

CORSO

JAVA EE

Indice generale

INTRODUZIONE A JAVA.....	8
JAVA PLATFORM.....	9
VANTAGGI E SVANTAGGI JAVA.....	9
JRE, JDK E AMBIENTI DI SVILUPPO.....	10
JAVA RUNTIME ENVIRONMENT (JRE).....	10
JAVA DEVELOPMENT KIT (JDK).....	10
AMBIENTI DI SVILUPPO CON INTERFACCIA GRAFICA.....	10
TIPI DI SOFTWARE.....	11
STANDALONE.....	11
CLIENT / SERVER.....	11
APPLICAZIONE WEB.....	11
SVILUPPO SOFTWARE: WATERFALL VS AGILE.....	12
METODOLOGIA WATERFALL.....	12
METODOLOGIA AGILE.....	13
COMPILAZIONE ED ESECUZIONE DEL PROGRAMMA.....	14
FILE JAR.....	14
FILE MANIFEST.....	14
DEBUG.....	15
VARIABILI E TIPI DI DATO.....	16
TIPI DI DATO.....	17
TIPI DI DATO PRIMITIVI.....	18
BOOLEAN.....	18
BYTE.....	18
SHORT.....	18
INT.....	18
LONG.....	18
FLOAT.....	19
DOUBLE.....	19
CHAR.....	19
LA CLASSE STRING.....	20
CONCATENAZIONE.....	20
TRASFORMAZIONE.....	21
SOSTITUZIONE.....	21
ESTRAZIONE.....	22
CONFRONTO.....	22
ALTRI METODI UTILI PER LE STRINGHE.....	23
OPERATORI.....	23
OPERATORE PUNTO (.).....	23
OPERATORI ARITMETICI.....	24
OPERATORI LOGICI.....	24
OPERATORI RELAZIONALI O DI CONFRONTO.....	25
CASTING.....	26
METODI.....	26
PARAMETRI.....	27
SIGNATURE.....	27
OVERLOADING DEI METODI.....	27
VARARGS.....	27
MODIFICATORE DI ACCESSO AI METODI.....	28
I MODIFICATORI (PIÙ IN GENERALE).....	29
MODIFICATORI DI ACCESSO.....	29

ALTRI MODIFICATORI: FINAL E STATIC.....	32
SINTASSI.....	33
CLASSI E INTERFACCIE.....	33
METODI E VARIABILI.....	34
PACKAGE.....	36
COMANDO IMPORT.....	36
NAMING E CODE CONVENTION.....	37
NAMING E CODE CONVENTION PER LE CLASSI.....	37
NAMING E CODE CONVENTION PER LE VARIABILI.....	37
NAMING CONVENTION PER I METODI.....	37
COMANDI CONDIZIONALI.....	38
IF – ELSE.....	38
SWITCH – CASE.....	39
COMANDO RETURN.....	40
CICLI.....	41
WHILE.....	41
DO – WHILE.....	41
FOR.....	41
ESEMPIO DI TUTTI E 3 I CICLI.....	42
COMANDI DI INTERRUZIONE DI CICLO.....	43
BREAK.....	43
CONTINUE.....	43
ESEMPIO CON BREAK E CONTINUE.....	44
PROGRAMMAZIONE ORIENTATA AGLI OGGETTI.....	45
INCAPSULAMENTO (parte 1).....	47
EREDITARIETÀ (parte 1).....	48
POLIMORFISMO (parte 1).....	48
CONSEGUENZE A LIVELLO HARDWARE DOPO AVER ISTANZIATO UNA CLASSE.....	49
RELAZIONE TRA CLASSI.....	50
ASSOCIAZIONE.....	51
ECCEZIONI.....	52
ECCEZIONI CHECKED.....	52
ECCEZIONI UNCHECKED.....	53
MECCANISMI DI GESTIONE DELLE ECCEZIONI.....	54
TRY – CATCH – FINALLY.....	59
THROW E THROWS.....	60
ARRAY, LISTE E COLLECTION, CLASSI WRAPPER, AUTOBOXING E UNBOXING.....	62
LISTE.....	62
MAPPE.....	63
ARRAY.....	64
ARRAY MULTIDIMENSIONALI.....	67
INTERFACCE COLLECTION E LIST.....	68
INTERFACCIA COLLECTION.....	68
INTERFACCIA LIST.....	69
CLASSI ARRAYLIST, HASHMAP E PROPERTIES.....	69
CLASSE ARRAYLIST.....	69
CLASSE HASHMAP.....	72
CLASSE PROPERTIES.....	74
INTERFACCIA ITERATOR.....	75
CICLARE LE COLLECTION.....	76
CLASSI WRAPPER.....	78
BOXING, AUTOBOXING E UNBOXING.....	79

INTERFACCE.....	80
CLASSI ASTRATTE.....	83
CLASSE INNER, LOCALE E ANONIMA.....	86
APPROFONDIMENTO CLASSI INNER.....	92
JAVA GENERICS.....	96
ANNOTATIONS.....	97
GESTIONE DEI FILE.....	99
INTRODUZIONE ALLA GESTIONE DEI FILE.....	99
CLASSE FILE.....	100
CLASSI PER SCRIVERE SU FILE.....	102
CLASSE FILEWRITER.....	102
CLASSE BUFFEREDWRITER.....	102
ESEMPIO CON LE CLASSI PER LA SCRITTURA DI UN FILE.....	102
CLASSI PER LEGGERE FILE.....	104
CLASSE FILEREADER.....	104
CLASSE BUFFEREDREADER.....	104
ESEMPIO CON LE CLASSI PER LA LETTURA DI UN FILE.....	104
CLASSE INPUTSTREAMREADER PER LEGGERE GLI INPUT DA TASTIERA.....	106
CLASSE SCANNER PER LEGGERE GLI INPUT DA TASTIERA.....	107
THREAD E CONCORRENZA.....	109
PROCESSO.....	109
THREAD.....	109
CONCORRENZA.....	111
CICLO DI VITA DI UN THREAD.....	112
THREAD PRIORITY.....	113
CREARE UN THREAD IN JAVA.....	114
MULTITHREADING.....	115
CONCORRENZA IN JAVA.....	118
CONCORRENZA CON L'UTILIZZO DEI THREAD.....	118
CONCORRENZA CON L'UTILIZZO DEGLI EXECUTOR.....	121
CONCORRENZA CON L'UTILIZZO DEL FRAMEWORK FORK/JOIN.....	123
SINCRONIZZAZIONE.....	125
ESEMPIO MULTITHREADING E CONCORRENZA : PRODUCER-CONSUMER.....	130
METODI WAIT(), NOTIFY(), NOTIFYALL().....	134
WAIT().....	134
NOTIFY() E NOTIFYALL().....	135
SINCRONIZZAZIONE AVANZATA CON LOCK E REENTRANTLOCK.....	135
LOCK.....	135
REENTRANTLOCK.....	136
UTILIZZARE IL BLOCCO TRY - FINALLY CON I THREAD.....	137
THREAD POOL.....	139
CLASSI ARRAYBLOCKINGQUEUE E LINKEDBLOCKINGQUEUE.....	142
CLASSE ARRAYBLOCKINGQUEUE.....	143
CLASSE LINKEDBLOCKINGQUEUE.....	145
GESTIONE BANCONE SALUMI MEDIANTE THREAD POOL.....	146
DEADLOCK, STARVATION E LIVELOCK.....	150
DEADLOCK.....	150
STARVATION.....	150
LIVELOCK.....	151
ESPRESSIONI LAMBDA.....	152
INTRODUZIONE ALLE ESPRESSIONI LAMBDA.....	152
SINTASSI DELLE ESPRESSIONI LAMBDA.....	154

PACKAGE JAVA.UTIL.FUNCTION.....	156
JAVA.UTIL.FUNCTION.PREDICATE.....	157
JAVA.UTIL.FUNCTION.CONSUMER.....	160
JAVA.UTIL.FUNCTION.SUPPLIER.....	161
JAVA.UTIL.FUNCTION.FUNCTION.....	162
JAVA.UTIL.FUNCTION.UNARYOPERATOR.....	162
JAVA.UTIL.FUNCTION.BINARYOPERATOR.....	163
ESEMPIO CASO D'USO CON LE LAMBDA: FILTRARE UNA LISTA IN BASE A DEI CRITERI.....	163
ESPRESSIONI REGOLARI.....	166
SINTASSI DELLE ESPRESSIONI REGOLARI.....	166
CLASSE PATTERN.....	171
CLASSE MATCHER.....	173
APPLICAZIONI PRATICHE DELLE ESPRESSIONI REGOLARI.....	176
VALIDAZIONE EMAIL.....	176
VALIDAZIONE FORMATO DATA.....	177
VALIDAZIONE CODICE FISCALE.....	178
LE DATE.....	179
INTRODUZIONE ALLA GESTIONE DELLE DATE IN JAVA.....	179
JAVA.SQL.TIMESTAMP.....	179
JAVA.UTIL.DATE.....	179
JAVA.UTIL.CALENDAR E JAVA.UTIL.GREGORIANCALENDAR.....	180
JAVA.TIME.LOCALDATE.....	181
JAVA.TIME.LOCALTIME.....	183
JAVA.TIME.LOCALDATETIME.....	184
JAVA.TIME.PERIOD E JAVA.TIME.DURATION.....	184
JAVA.TEXT.SIMPLEDATEFORMAT.....	185
DATABASE.....	187
INTRODUZIONE AI DATABASE.....	187
OPERAZIONI CRUD.....	187
INTERFACCIARSI CON I DATABASE IN JAVA – JDBC.....	189
OPERAZIONI CRUD CON IL JDBC CONNECTOR.....	190
SELECT CON JDBC.....	190
INSERT, UPDATE E DELETE CON JDBC.....	192
PREVENIRE ATTACCHI SQL INJECTION CON I PREPAREDSTATEMENT.....	195
LIBRERIE GRAFICHE IN JAVA.....	198
COMPONENTE GUI.....	198
AWT.....	199
SWING.....	200
SWT.....	201
TESTING DEL SOFTWARE CON JUNIT.....	202
INTRODUZIONE AGLI UNIT TEST.....	202
INTRODUZIONE AL FRAMEWORK JUNIT.....	203
CLASSE ASSERT.....	206
UNIT TEST PARAMETRIZZATO.....	208
STORIA DEL WEB.....	211
APPLICAZIONI CLIENT / SERVER.....	212
STANDARD WEB.....	212
PROTOCOLLI DI RETE.....	213
PILA PROTOCOLLARE TCP / IP.....	213
PROTOCOLLO TCP.....	214
PROTOCOLLO UDP.....	215
PORTE DEL PROTOCOLLO TCP / IP.....	215

PROTOCOLLO HTTP.....	216
WEB APPLICATION VS SITO STATICO.....	218
ELABORAZIONE DINAMICA DELLA RISPOSTA.....	219
CLIENT WEB.....	219
SERVER WEB.....	219
DALLA RICHIESTA ALLA RISPOSTA.....	220
LINGUAGGI PER L'ELABORAZIONE DI WEB APPLICATION.....	221
JEE.....	222
APACHE TOMEET.....	224
INSTALLAZIONE JDK.....	224
INSTALLAZIONE APACHE TOMEET.....	228
COMPONENTI DI APACHE TOMEET.....	229
WEB APPLICATION.....	232
ANATOMIA DI UNA WEB APPLICATION.....	232
PRESENTATION LAYER.....	232
BUSINESS LAYER.....	232
DATA LAYER.....	233
PATTERN MODEL-VIEW-CONTROLLER (MVC).....	233
SERVLET.....	235
CICLO DI VITA DI UNA SERVLET.....	236
HTTPSERVLETREQUEST E HTTPSERVLETRESPONSE.....	238
PASSAGGIO DI PARAMETRI AD UNA SERVLET.....	243
REQUEST DISPATCHING.....	247
HTTPSESSION.....	252
JAVA FILTERS E FILTER CHAIN.....	257
PRE-PROCESSING DELLE RICHIESTE.....	257
POST-PROCESSING DELLE RISPOSTE.....	258
IMPLEMENTAZIONE DI UN FILTRO.....	258
FILTER CHAIN.....	262
SERVLETLISTENERS.....	268
SERVLET CHE GENERA PDF CON LA LIBRERIA ITEXT.....	270
JAVA SERVER PAGES (JSP).....	272
SCRIPTLET.....	272
DICHIARAZIONI.....	273
ESPRESSIONI.....	273
DIRETTIVE.....	274
AZIONI.....	276
OGGETTI IMPLICITI: REQUEST, RESPONSE, OUT, SESSION, APPLICATION.....	281
TAG LIBRARY.....	283
TAG SENZA CORPO.....	284
TAG CON CORPO.....	284
FILE TDL (TAG LIBRARY DESCRIPTOR).....	284
UTILIZZO DELLE TAG LIBRARY NELLE JSP.....	285
JSTL – JSP STANDARD TAG LIBRARY.....	285
CONFIGURAZIONE DELLA WEB APP PER L'UTILIZZO DI JSTL.....	286
TAG CORE.....	287
TAG FORMAT.....	293
ENTERPRISE JAVA BEAN (EJB).....	296
DEPENDENCY INJECTION.....	297
EJB STATELESS.....	298
METODI ASINCRONI IN UN EJB.....	302
GESTIONE DELLE TRANSAZIONI NEGLI EJB CONTAINER-MANAGED.....	303

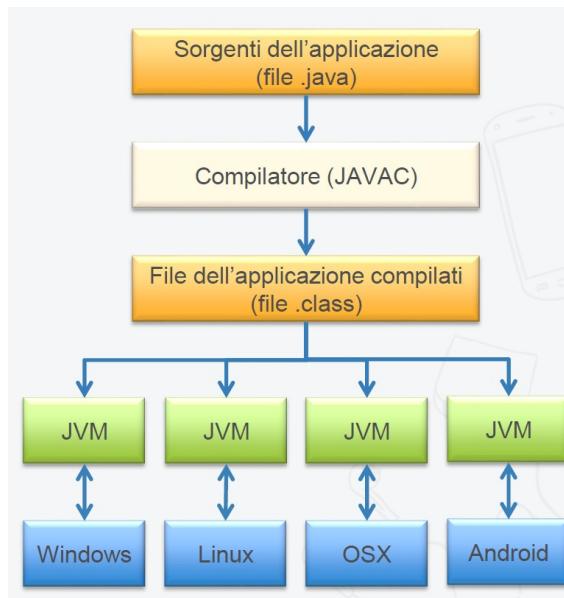
EJB STATEFUL.....	303
QUANDO USARE EJB STATELESS O EJB STATEFUL.....	309
INTERCEPTOR.....	309
MESSAGE DRIVEN BEAN (MDB).....	311
WEB SERVICE SOAP CON EJB.....	312
WEB SERVICE RESTful CON EJB.....	315

INTRODUZIONE A JAVA

JAVA è un linguaggio di programmazione object oriented (orientato agli oggetti). La programmazione a oggetti è un paradigma ancora oggi molto utilizzato, perché ci consente di modellare al meglio delle situazioni reali.

La sintassi del linguaggio JAVA è molto simile ai linguaggi C e C++, da cui eredita parecchie caratteristiche. Per certi versi, però, JAVA è più semplice da utilizzare rispetto i due linguaggi citati precedentemente.

La caratteristica principale che ha reso JAVA così popolare è la portabilità, cioè l'essere indipendente dal sistema operativo su cui viene eseguito un software. Gli elementi fondamentali che rendono JAVA un linguaggio di programmazione portabile sono la Java Virtual Machine (Macchina virtuale o JVM) e la Java Platform.



Appunto, portabilità vuol dire scrivere il proprio codice sorgente una sola volta, compilarlo ed eseguirlo su qualsiasi dispositivo dotato di una macchina virtuale. Un programma JAVA è rappresentato da uno o più file.java, al cui interno sono presenti i nostri sorgenti, ovvero il codice che scriviamo. Attraverso il compilatore chiamato JAVAC, il nostro codice sorgente viene tradotto in un linguaggio intermedio, chiamato bytecode. Tutti questi file compilati avranno l'estensione .class e potranno essere interpretati dalla JVM, che è un processore virtuale che legge il/i nostro/i file.class e traduce il bytecode al loro interno in linguaggio macchina. Quindi, se su dispositivi differenti, con sistema operativo differente, abbiamo la stessa JVM, possiamo eseguire il nostro software senza necessità di ricompilarlo ogni volta.

Oggi JAVA è utilizzato per scrivere:

- applicazioni web: La maggior parte delle applicazioni enterprise (applicazioni aziendali) utilizzano JAVA, soprattutto nella parte backend;
- applicazioni per smartphone e tablet: Android, per esempio, è un sistema operativo scritto in JAVA, così come tutte le app scritte per questo sistema operativo;
- applicazioni per decoder digitali, elettrodomestici e così via.

JAVA PLATFORM

La Java platform è una piattaforma composta da due componenti:

- JVM, di cui abbiamo già parlato precedentemente;
- API (Application Programming Interface), cioè un set di librerie (componenti software) messi a disposizione degli sviluppatori per poter scrivere software JAVA.

La Java Platform è disponibile in 3 configurazioni:

- Java Standard Edition (JSE): mette a disposizione il set standard di API per poter scrivere applicazioni standalone (programma che può funzionare senza che siano richiesti altri componenti o addirittura senza sistema operativo), client e server, per accesso a database, per il calcolo scientifico e così via;
- Java Enterprise Edition (JEE): mette a disposizione, oltre alle API della Standard Edition, anche quelle per scrivere applicazioni distribuite (ad esempio applicazioni web);
- Java Micro Edition (JME): mette a disposizione le API per sviluppare applicazioni mobile.

VANTAGGI E SVANTAGGI JAVA

Vantaggi

- indipendenza del linguaggio bytecode: consente di eseguire lo stesso programma su più dispositivi dotati di JVM;
- velocità di sviluppo;
- grande disponibilità di librerie;
- alta integrazione con il web.

Svantaggi

- velocità di esecuzione: il programma viene eseguito ed elaborato dalla JVM, che a sua volta traduce le istruzioni in linguaggio macchina. Pertanto il tempo di esecuzione è leggermente più lento rispetto ad un programma scritto in C++;
- attraverso la decompilazione è possibile risalire al codice sorgente.

JRE, JDK E AMBIENTI DI SVILUPPO

JAVA RUNTIME ENVIRONMENT (JRE)

JRE è un'implementazione della JVM, ed è necessario per l'esecuzione dei programmi JAVA.

Il JRE contiene:

- JVM.
- API standard di JAVA.
- Un launcher necessario per avviare i programmi già compilati in bytecode. Tutti i programmi partono sempre da una classe dotata di un metodo main()

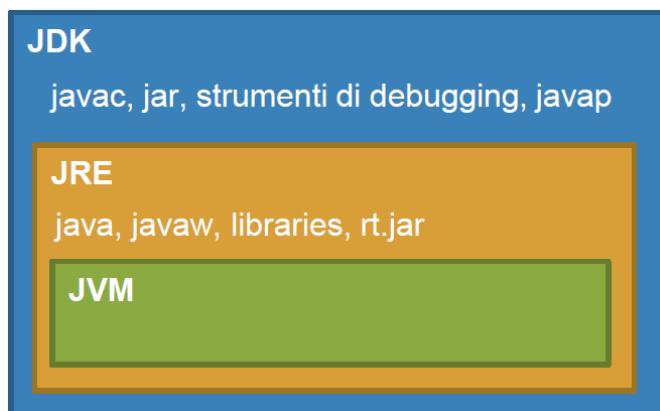
Il JRE deve essere installato su tutti i dispositivi che hanno necessità di eseguire software scritti in JAVA.

JAVA DEVELOPMENT KIT (JDK)

JDK è un insieme di librerie e software messe a disposizione da Oracle, che consentono di sviluppare software scritti in JAVA.

JDK è un ambiente di sviluppo a tutti gli effetti, poiché appunto contiene tutte queste librerie che ci consentono di sviluppare software, ma è un ambiente di sviluppo a console, ovvero non è dotato di un'interfaccia grafica, ma le istruzioni si eseguono mediante il prompt di comandi. Gli ambienti di sviluppo più famosi, dotati di interfaccia grafica, sono Eclipse, Netbeans e IntelliJ IDEA.

Il JDK contiene librerie importanti per lo sviluppo, il JRE e la JVM.



AMBIENTI DI SVILUPPO CON INTERFACCIA GRAFICA

Un ambiente di sviluppo integrato o IDE (Integrated Development Environment), rispetto il JDK, è un software dotato di interfaccia grafica che consente agli sviluppatori di creare software JAVA, semplificando la programmazione e la gestione dei file. È utile perché:

- consente di segnalare e visualizzare subito gli errori di sintassi all'interno del codice;
- consente di effettuare il debug in maniera semplice;
- offre una serie di strumenti e funzionalità di supporto allo sviluppatore.

TIPI DI SOFTWARE

STANDALONE

Il software standalone è un software che è installato all'interno di un sistema operativo ed è autonomo, perché non richiede l'uso di particolari componenti esterne con cui interagire (per esempio i server).

Esempi di software standalone sono:

- il pacchetto di software Microsoft (Word, Excel, ...);
- i software Adobe (Photoshop, InDesign, ...);
- alcune app per dispositivi mobile (ad es. l'app che visualizza le foto, l'app che gestisce i file del device).

CLIENT / SERVER

Il software client/server è composto dai due componenti omonimi che definiscono il suo nome. La componente client è installata sul nostro dispositivo personale, mentre la componente server si trova su un dispositivo remoto e fornisce un servizio al client.

Esempi di sistemi client/server:

- File server: per la condividere i file;
- FTP server: per l'upload/download dei file;
- Database server: per la gestione dei dati;

Questa architettura, però, ha un limite importante. Questo limite riguarda la necessità di installare il software della componente client su ciascun terminale che deve accedere a questo software. Questo problema si risolve con le applicazioni web.

APPLICAZIONE WEB

L'applicazione web è nata con lo scopo di migliorare l'utilizzo dei componenti client, senza la necessità di dover installare nuovi software. Un'applicazione web è accessibile mediante un normale browser.

L'applicazione web è un tipo di client/server evoluto, in cui è sostanzialmente il browser a fare da client, che scarica per noi tutte le varie informazioni (le pagine web) che ci servono per interagire con la componente server. In questo caso, il server e il client comunicano mediante protocollo HTTP.

Per accedere alle applicazioni web si utilizzano URL o link ipertestuali.

Sviluppo Software: Waterfall vs Agile

Dopo aver visto i vari tipi di software, adesso vediamo come si sviluppa un software. Non ha senso parlare di sviluppo nel momento in cui abbiamo a che fare con degli script o piccoli software che devono svolgere un compito particolare, ma solo nel momento in cui abbiamo a che fare con software complessi o strutturati per la gestione di un'intera azienda.

Usare una metodologia per lo sviluppo di un software complesso è importante, perché consente di realizzare il software in maniera più organizzata e strutturata, limitando gli errori che uscirebbero fuori nel caso di uno sviluppo senza criterio.

Le metodologie per lo sviluppo di un software sono 2: waterfall e agile.

METODOLOGIA WATERFALL

Nella metodologia waterfall (o classica) la sequenza delle fasi del ciclo di vita di un progetto software è strettamente sequenziale e, prima di passare alla fase successiva, è necessario che quella corrente sia terminata completamente.

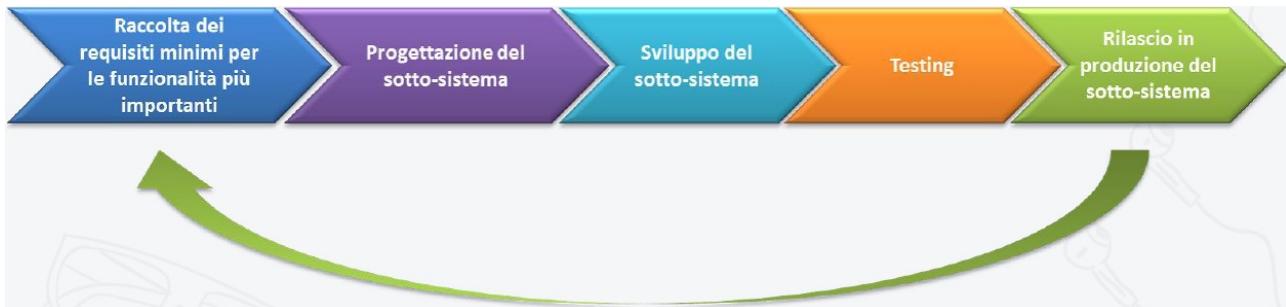


Quindi, in questo tipo di approccio raccogliamo i requisiti all'inizio, sulla base di quei requisiti viene sviluppato il software e, infine, facciamo vedere il nostro software completo solo al termine del rilascio in produzione.

Il limite principale di questa metodologia è quello di aver realizzato un prodotto che soddisfa perfettamente i requisiti inizialmente raccolti che, però, nel frattempo possono essere cambiati e quindi il software è diverso da quello atteso.

METODOLOGIA AGILE

Nella metodologia agile abbiamo le stesse fasi della metodologia waterfall, ma il ciclo di vita di un progetto software è visto come una sequenza di tante iterazioni, dove ogni iterazione è un arco temporale che va dalle 2 alle 4 settimane.



In ciascuna iterazione prendiamo un pezzettino del nostro software sul quale vengono eseguite tutte le fasi, dopodiché si prende un altro pezzettino e si ritorna indietro nelle fasi per rieseguirle tutte e così via, fino alla realizzazione di tutti i pezzettini che comporranno il software finale.

Vantaggi di questa metodologia:

- coinvolgimento attivo del committente e degli utenti nel processo di sviluppo;
- aggiornamenti regolari e frequenti sullo stato dell'applicazione;
- validazione continua dei requisiti (dopo ogni iterazione);
- consegna rapida delle funzionalità di base;
- pianificazione fissa dei tempi di consegna per funzionalità;
- maggiori test, software migliore.

COMPILAZIONE ED ESECUZIONE DEL PROGRAMMA

La compilazione è quel processo che consente di trasformare il codice sorgente, che si trova all'interno dei nostri file.java, in un linguaggio intermedio chiamato bytecode, che è il linguaggio interpretato dalla JVM.

Tutti i file compilati si trovano all'interno di file che hanno estensione .class . Quindi, a partire dal nostro sorgente .java, tramite la compilazione viene creato uno o più file .class .

Il tool messo a disposizione dalla JDK per compilare il nostro software è il tool javac.

FILE JAR

Un JAR è un file compresso (archivio o zip) che contiene al suo interno una serie di file compilati del nostro software, ed eventuali librerie aggiuntive. I vantaggi dell'uso del JAR sono:

- Compressione, perché i file vengono compressi in un unico file, quindi la dimensione complessiva del programma diminuisce.
- Firma, perché il file JAR contiene una cartella META-INF, che contiene a sua volta il file MANIFEST.MF. Questo file consente di identificare al meglio il nostro software, ossia identificare per esempio il Class-Path, il Main-Class, l'autore e così via.
- Protabilità, perché lo stesso JAR può essere eseguito su diversi sistemi operativi che contengono la JVM richiesta per l'esecuzione.

FILE MANIFEST

Alcuni attributi del file MANIFEST sono:

- Manifest-Version: definisce appunto la versione del file MANIFEST.
- Created-By: definisce la versione e il vendor del JDK utilizzato per creare il file.
- Class-Path: definisce il percorso delle librerie che sono necessarie per eseguire il software.
- Main-Class: contiene la classe che ha il metodo main() per avviare il nostro software.

DEBUG

Il debug è il miglior strumento che un programmatore può utilizzare durante lo sviluppo di un software. Questo strumento serve sostanzialmente per testare il codice che abbiamo scritto, prima di effettuare altre operazioni. In particolare, il debug:

- serve per osservare il comportamento del nostro programma quando è in fase di esecuzione e di individuare in anticipo gli eventuali errori logici e di compilazione, che sono nascosti e mascherati all'interno del nostro software;
- è un'attività che consiste nell'eseguire il programma a step, interrompendo l'esecuzione ad una certa istruzione;

Il debug può essere fatto in due modi:

- debug tramite il log, dove il log è uno strumento che scrive delle istruzioni all'interno di un file, o all'interno della console. La sintassi generica è la seguente:
`log.debug("Questa frase uscirà nel log solo in fase di debug del software");`
- debug tramite IDE, che si effettua inserendo dei breakpoint (punti di interruzione) all'interno del nostro codice sorgente. I breakpoint consentono di interrompere l'esecuzione del programma e di eseguire, da quel punto in poi, le istruzioni passo passo.

VARIABILI E TIPI DI DATO

Consideriamo che un software, per essere eseguito, ha bisogno del processore che svolge i calcoli e della memoria al cui interno vengono salvate le variabili che utilizziamo durante l'esecuzione del nostro programma. Una variabile è una porzione di questa memoria, che viene riservata al nostro programma e al cui interno salviamo i nostri dati di volta in volta, man mano che utilizziamo la variabile.

Il nome della variabile rappresenta l'indirizzo fisico in cui è presente la variabile e serve per indicare, all'interno della memoria, il punto in cui è presente la nostra variabile, in modo da potervi accedere in lettura per visualizzarne il valore, o in scrittura per modificarlo.

Una variabile assume un preciso significato in base al punto di codice in cui viene definita. In JAVA ci sono 4 tipi di variabili:

- le variabili locali sono definite all'interno di un metodo. Sono create quando il metodo viene invocato e cancellate dalla memoria quando il metodo viene terminato;
- le variabili di istanza sono definite all'interno di una classe, ma fuori dai metodi della classe stessa (altrimenti diventano variabili locali);
- le variabili di classe sono variabili di istanza che hanno il modificatore static;
- i parametri sono variabili che vengono dichiarate all'interno delle parentesi tonde di un determinato metodo.

Facciamo un esempio:

```
public class Variabili {  
    public int b = 5; /*variabile d'istanza*/  
  
    public static String stampa = "Ciao"; /*variabile di classe*/  
  
    public static void main(String[] args) { /*La variabile args è un parametro*/  
  
        Variabili c = new Variabili();  
        c.b = 10; /*richiamo e modifica della variabile d'istanza, che richiede  
                   la creazione di un oggetto con new*/  
  
        System.out.println(Variabili.stampa); /*richiamo della variabile di classe,  
che non richiede la creazione di un oggetto con new*/  
  
        int a = 10; /*variabile Locale*/  
    }  
}
```

Una variabile, per essere utilizzata, deve essere prima dichiarata e poi inizializzata. La dichiarazione serve per riservare uno spazio di memoria per la nostra variabile. Ovviamente, quando viene dichiarata, la variabile non ha valore. L'inizializzazione invece è l'operazione che consente di assegnare il valore alla variabile che abbiamo dichiarato. Posso dichiarare una variabile senza inizializzarla, ma non posso inizializzare una variabile se non l'ho dichiarata.

Vediamo un esempio:

```
public class Variabili {  
    public static void main(String[] args) { /*La variabile args è un parametro*/  
        int a = 10; /*variabile locale*/  
        Variabili v; /*dichiarazione*/  
        v = new Variabili();/*inizializzazione*/  
        Variabili v2 = new Variabili();/*dichiarazione e inizializzazione*/  
    }  
}
```

TIPI DI DATO

Il tipo di dato è un insieme di caratteristiche. Una variabile che ha un determinato tipo di dato, può avere solo valori che soddisfano le caratteristiche del tipo di dato per cui è stata definita. Per esempio, se definisco una variabile di tipo int, tale variabile può contenere solo valori di tipo intero, altrimenti avrò errori già in fasi di compilazione.

In JAVA esistono due tipi di dato:

- i tipi di dato primitivi, che sono 8 (boolean, byte, char, double, float, int, long, short) ed utilizzano una quantità di memoria predefinita. Ogni tipo primitivo ha un valore di default;
- i tipi di riferimento, (classi, interfacce,...) che utilizzano ovviamente una quantità di memoria che varia in funzione del numero di informazioni contenute. Più informazioni ci sono in un tipo di riferimento in una classe, tanto più sarà la memoria richiesta per poter istanziare un oggetto di quel tipo. Ogni variabile di questo tipo viene inizializzata di default con il valore null.

TIPI DI DATO PRIMITIVI

I tipi di dato primitivi sono 8: boolean, byte, char, double, float, int, long e short.

BOOLEAN

Il tipo di dato boolean rappresenta i valori vero o falso. Non è specificata quanta memoria utilizza, ma basterebbe comunque un solo bit per specificare true o false. Il valore di default è false.

Generalmente questa variabile boolean si utilizza all'interno dello statement if.

BYTE

Il tipo di dato byte rappresenta i valori interi che vanno da -128 a 127 (inclusi). Questo tipo di dato utilizza 8 bit di memoria e viene usato generalmente per risparmiare della memoria in array di grandi dimensioni.

Nella variabile di tipo byte non sono ammesse le operazioni aritmetiche, quindi somma, sottrazione, moltiplicazione e divisione.

Esempio

```
byte a = 100;  
byte b = -5;  
byte c = (byte) (a+b);  
  
/* nella console viene stampato 95 */  
System.out.println(c);
```

NOTA: la variabile **c** contiene il
valore in byte della somma tra **a** e **b**.
a+b = 95 quindi il valore di **c** è 95.

```
byte d = 100;  
byte e = 50;  
byte f = (byte) (d+e);  
  
/* nella console viene stampato -106 */  
System.out.println(f);
```

NOTA: la variabile **f** contiene il
valore in byte della somma tra **d** e **e**.
d+e = 150 quindi il valore di **f** è -106.

Esempio

```
String str = "frase di esempio";  
/* getBytes() codifica la stringa in una sequenza di byte e li  
salva in un array di byte */  
byte[] strByte = str.getBytes();  
for(int i = 0; i < strByte.length; i++) {  
/* stampiamo il valore in byte di ogni carattere della stringa */  
System.out.print(strByte[i] + " ");  
}
```

L'output nella console sarà: 102 114 97 115 101 32
100 105 32 101 115 101 109 112 105 111
102 è il valore in byte della lettera **f**

SHORT

Il tipo di dato short rappresenta valori interi compresi tra -32768 e 32767 (inclusi). Una variabile short occupa 16 bit e, anche in questo caso, non sono ammesse le operazioni aritmetiche.

INT

Il tipo di dato int rappresenta valori interi compresi tra -2147483648 e 2147483647 (inclusi). È il tipo di dato utilizzato per rappresentare i valori interi. Su questo tipo di variabile è possibile effettuare le operazioni aritmetiche.

LONG

Il tipo di dato long è come il tipo di dato int, solo che occupa 64 bit. Questo tipo di dato è utilizzato quando si lavora, ovviamente, con numeri di grandi dimensioni e le operazioni aritmetiche sono ammesse anche in questo caso.

FLOAT

Il tipo di dato float rappresenta i numeri in virgola mobile, cioè numeri reali con precisione singola. Il tipo float utilizza 32 bit e il valore di default 0.0f. Questo tipo di dato è consigliabile non utilizzarlo quando vogliamo ottenere valori precisi, ad esempio quando si lavora con le valute. Quando si lavora su valori precisi esiste la classe java.math.BigDecimal.

DOUBLE

Il tipo di dato double rappresenta i numeri in virgola mobile, cioè sempre numeri reali, ma con precisione doppia stavolta. I double occupano 64 bit e il valore di default è 0.0d. Quando si lavora con numeri decimali, il consiglio è quello di usare il tipo di dato double rispetto al float.

Vediamo un esempio con float e double:

```
public class FloatDouble {  
  
    public static void main(String[] args) {  
  
        float a = 100.45f; /* oppure float a = (float) 100.45*/  
        float b = -10.30f; /* oppure float b = (float) -10.30*/  
  
        System.out.println("La somma di tipo float e' " + (a + b)); //La somma di tipo  
        float e' 90.149994  
  
        double c = 100.45;  
        double d = -10.30;  
  
        System.out.println("La somma di tipo double e' " + (c + d)); //La somma di tipo  
        double e' 90.15  
    }  
}
```

CHAR

Il tipo di dato char rappresenta un carattere Unicode. Unicode è il sistema di codifica che assegna un numero univoco ad ogni carattere utilizzato per la scrittura di testi. Il valore più piccolo è '\u0000' (cioè 0), mentre il valore più grande è '\uffff' (cioè 65535). Il char occupa 16 bit e il valore di deaful è '\u0000' (cioè 0).

LA CLASSE STRING

La classe String è la classe fondamentale di JAVA che ci consente di lavorare e manipolare le stringhe.

Una stringa è una sequenza finita di caratteri racchiusa tra virgolette, dove per carattere si intendono lettere, numeri, apici, apostrofi, caratteri speciali e così via.

In JAVA le stringhe sono rappresentate dall'oggetto String, che si trova all'interno del package java.lang. Da fare attenzione sul fatto che JAVA non ha il tipo di dato primitivo string.

Facciamo un esempio:

```
String val1 = "Lorem ipsum...";
```

La classe String ha una particolarità, ossia è una classe immutabile. Essendo tale classe immutabile, non può essere estesa, quindi non possiamo creare una nostra classe che eredita la classe String.

Per ottenere un'istanza della classe String ci sono 4 modi:

- Modo 1 - Stringa con valore

```
String str1 = "Stringa 1";
```

- Modo 2 - Stringa con valore nullo

```
String str2;
```

- Modo 3 - Stringa con creazione ed inizializzazione dell'oggetto

```
String str3 = new String("Stringa 3");
```

- Modo 4 - Stringa creata a partire da un array di char

```
char[] array = {'S', 't', 'r', 'i', 'n', 'g', ' ', 'a', ' ', '3'};  
String str4 = new String(array);
```

CONCATENAZIONE

Concatenare due o più stringhe vuol dire unirle tutte in un'unica stringa.

La concatenazione si può fare in due modi:

- con l'operatore +;
- con il metodo concat() messo a disposizione dalla classe String.

Per esempio:

```
public class Stringhe {  
    public static void main(String[] args) {  
        String val1 = "Lorem ipsum...";  
        String val2 = "test,,,,";  
  
        String val3 = val1 + val2;  
        System.out.println(val3); //Lorem ipsum...test,,,  
  
        String val4 = val1.concat(val2);  
        System.out.println(val4); //Lorem ipsum...test,,,  
    }  
}
```

TRASFORMAZIONE

La trasformazione consente di modificare la stringa in ingresso, cambiando l'aspetto dei suoi caratteri. I metodi messi a disposizione dalla classe String per la trasformazione di una stringa sono `toLowerCase()`, `toUpperCase()` e `trim()`.

Per esempio:

Esempi: Trasformazione in minuscolo

```
String str1 = " StringA 1 ";
System.out.println(str1.toLowerCase());
```

Output: stringa 1

Esempi: Trasformazione in maiuscolo

```
String str2 = " StringA 2 ";
System.out.println(str2.toUpperCase());
```

Output: STRINGA 2

Esempi: Rimozione di spazi iniziali e finali

```
String str3 = " StringA 3 ";
String str4 = str3.trim();
```

Il valore di str4 è "StringA 3" (senza spazi iniziali e finali)

SOSTITUZIONE

La sostituzione consente di sostituire uno o più caratteri all'interno di una stringa. I metodi messi a disposizione dalla classe String per la sostituzione di caratteri nelle stringhe sono:

- `replace(CharSequence target, CharSequence replacement)`: sostituisce tutto quello che fa parte di target con la sequenza di caratteri che troviamo all'interno di replacement;
- `replaceAll(String regex, String replacement)`: sostituisce tutte le sottostringhe che corrispondono all'espressione regolare regex con la sottostringa replacement;
- `replaceFirst(String regex, String replacement)`: sostituisce solo la prima sottostringa che corrisponde all'espressione regolare regex con la sottostringa replacement.

Per esempio:

```
public class Stringhe {
    public static void main(String[] args) {
        String val5 = "Questo e' il corso Java AVANZATO";
        /* replace */
        String val6 = val5.replace("a", "!");
        System.out.println(val6); //Questo e' il corso J!v! AVANZATO
        /* replaceAll*/
        String val7 = val5.replaceAll("[a-n]+", "4");
        /*tutte le lettere dalla a alla n minuscole sono sostituite dal 4*/
        System.out.println(val7); //Qu4sto 4' 4 4orso J4v4 AVANZATO
    }
}
```

```

    /* replaceFirst*/
    String val8 = val5.replaceFirst("[a-n]+", "P");
/*sostituisce con P solo la prima lettera trovata nella stringa tra a e n minuscola*/

    System.out.println(val8); //QuPsto e' il corso Java AVANZATO

}
}

```

ESTRAZIONE

L'estrazione consente di estrarre una sottostringa da una stringa. I metodi messi a disposizione dalla classe String per l'estrazione di caratteri dalle stringhe sono:

- `substring(beginIndex)`: specifica solamente l'indice di inizio estrazione;
- `substring(beginIndex, endIndex)`: specifica sia l'indice di inizio che l'indice di fine estrazione.

Per esempio:

```

public class Stringhe {

    public static void main(String[] args) {

        String val5 = "Questo e' il corso Java AVANZATO";

        /*substring*/
        String val9 = val5.substring(5);
        System.out.println(val9); //o e' il corso Java AVANZATO

        String val10 = val5.substring(0, 5);
        System.out.println(val10); //Quest
    }
}

```

CONFRONTO

Il confronto consente di comparare due stringhe. I metodi messi a disposizione dalla classe String per il confronto di due stringhe sono:

- `equals(Object anObject)`: effettua un confronto tra due stringhe e ritorna true se sono uguali, altrimenti ritorna false. Questo metodo è case sensitive, cioè tiene conto se un carattere è maiuscolo o minuscolo.
- `equalsIgnoreCase(String anotherString)`: svolge lo stesso compito del primo metodo, ma non è case sensitive, cioè non tiene conto se un carattere è maiuscolo e minuscolo.

Per esempio:

```

public class Stringhe {

    public static void main(String[] args) {
        /*equals*/
        String val11 = "Paolo Preite";
        String val12 = "paolo preite";
        System.out.println(val11.equals(val12)); //false
    }
}

```

```

        System.out.println(val11.equalsIgnoreCase(val12));//true
    }
}

```

ALTRI METODI UTILI PER LE STRINGHE

- `String.valueOf(int i)`: converte il numero intero in ingresso in una stringa. È possibile convertire anche tutti gli altri tipi di dato;
- `split(String regex)`: consente di dividere una stringa in un array di stringhe, secondo l'espressione regolare passata in ingresso;
- `startsWith(String prefix)`: ritorna true se la stringa inizia con il prefisso indicato, altrimenti false;
- `endsWith(String suffix)`: ritorna true se la stringa finisce con il suffisso indicato, altrimenti false;
- `charAt(int index)`: prende in ingresso un indice e ritorna il carattere posizionato su quell'indice

Per esempio:

```

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

        /*equals*/
        String val11 = "Paolo Preite";
        String val12 = "paolo preite";
        System.out.println(val11.equals(val12));//false
        System.out.println(val11.equalsIgnoreCase(val12));//true

        /*split*/
        String[] array = val11.split(" ");
        for(int i = 0; i < array.length; i++){
            System.out.println(array[i]);//Paolo
                                //Preite
        }

        /*startsWith e endsWith*/
        System.out.println(val11.startsWith("Pao")); //true
        System.out.println(val11.endsWith("Pao")); //false

        /*charAt*/
        System.out.println(val11.charAt(3));//L
    }
}

```

OPERATORI

OPERATORE PUNTO (.)

L'operatore punto ci consente di accedere alle variabili e metodi di una classe e alle variabili e metodi di un oggetto.

OPERATORI ARITMETICI

Tra gli operatori aritmetici abbiamo:

- somma con il simbolo + ;
- sottrazione con il simbolo - ;
- moltiplicazione con il simbolo * ;
- divisione con il simbolo / ;
- modulo con il simbolo %, per calcolare il resto di una divisione tra interi.

Tutti gli operatori aritmetici si possono esprimere direttamente con il loro simbolo, oppure con il simbolo seguito dall'uguale (`+=`, `-=`, `*=`, `/=`, `%=`). La differenza è che nel primo caso assegniamo il risultato dell'operazione ad una nuova variabile, mentre nel secondo caso assegniamo ad una variabile il valore che già possiede, modificato però dall'operazione e il numero che seguono nell'istruzione.

Per esempio:

```
public class OperatoriAritmetici {  
  
    public static void main(String[] args) {  
        int a = 10;  
        int b = 20;  
  
        int c = a + b;  
  
        System.out.println(c); // 30  
        System.out.println(a += b); // 30  
  
    }  
}
```

Consideriamo che il tipo di dato ritornato dalle operazioni varia in funzione del tipo di dati degli operandi.

OPERATORI LOGICI

Gli operatori logici sono utilizzati per confrontare due elementi e ritornano come risultato due valori:

- vero (o true);
- falso (o false);

Gli operatori logici sono:

- AND (A `&&` B): restituisce true se e solo se A e B sono vere entrambe, altrimenti false;
- OR (A `||` B): restituisce true se e solo se almeno uno tra A o B è vero, altrimenti false;
- NOT (!A): se A è true, il NOT restituisce false e viceversa.

OPERATORI RELAZIONALI O DI CONFRONTO

Gli operatori relazionali confrontano due operandi, ritornando come risultato due valori:

- vero (o true);
- falso (o false);

Gli operatori relazionali sono:

- Maggiore di ($A > B$)
- Maggiore o uguale di ($A \geq B$)
- Minore di ($A < B$)
- Minore o uguale di ($A \leq B$)
- Uguale a ($A == B$)
- Diverso da ($A != B$)

Per ciascun operatore abbiamo true se il confronto risulta effettivamente essere veritiero, altrimenti false.

CASTING

Il casting è l'operazione che consente il passaggio di una variabile da un tipo di dato ad un altro. Questo vuol dire che posso convertire una variabile di un determinato tipo in una variabile di un altro tipo.

Il cast può essere:

- implicito, quando non abbiamo una dichiarazione esplicita;
- esplicito, quando la dichiarazione del casting viene proprio esplicitata attraverso il nostro codice.

Il cast esplicito si scrive attraverso la dichiarazione del nuovo tipo tra parentesi tonde, accanto alla variabile. Per esempio, (int) myvar converte il valore della variabile my var di tipo long in un int.

METODI

I metodi sono dei blocchi di codice che possono essere riutilizzati all'interno del nostro software. Essi vengono utilizzati per rappresentare il comportamento di classi e oggetti.

Per esempio, immaginiamo di avere una classe Persona, un metodo che rappresenta un'azione potrebbe essere cammina(), oppure il metodo mangia() e così via. Questi metodi effettuano al loro interno delle operazioni che simulano l'essere umano che cammina, o che mangia.

I metodi sono importanti perché, se abbiamo più volte bisogno di un blocco di istruzioni durante l'implementazione di un software, ci evitano di riscrivere sempre la stessa porzione di codice. Basterà, quindi, invocare il metodo di cui abbiamo bisogno, passando gli eventuali parametri in ingresso.

Un metodo si definisce nel seguente modo:

```
[modificatore di accesso] [altri modificatori] TipoRitornato nomeMetodo(parametri) [throws Eccezioni] {  
    // blocco di codice appartenente al metodo  
    return variabile;  
}
```

- [modificatore di accesso] = uno a scelta tra private, protected, public, default
- [altri modificatori] = static final (è possibile che siano presenti anche tutti e due)
- TipoRitornato = void se il metodo non ritorna niente (in questo caso lo statement return non va inserito), tipo primitivo o classe in base al valore ritornato
- nomeMetodo = nome del metodo scelto
- parametri = lista dei parametri ammessi in ingresso: Tipo1 param1,..., Tipo N paramN. È possibile anche definire metodi che non accettano parametri in ingresso
- throws = lista delle classi delegate alla gestione delle eccezioni in caso di errore ClasseEccezione1, ..., ClasseEccezioneN

PARAMETRI

I parametri in ingresso di un metodo possono essere di vario tipo e possono essere utilizzati all'interno del metodo.

```
public String getUserInfo(String nome, String cognome, int anni) {  
    return nome + " " + cognome + ", età: " + anni;  
}
```

Le variabili nome, cognome e anni possono essere utilizzate nel corpo del metodo (la parte tra parentesi graffe).

SIGNATURE

La firma o signature del metodo è data dalla coppia nome e parametri. Il nome da solo non dice niente.

Possiamo creare più metodi che hanno lo stesso nome, purché i parametri siano diversi.

OVERLOADING DEI METODI

L'overloading dei metodi è una caratteristica di JAVA, che consente di definire più volte il metodo in una classe, ovviamente utilizzando diversi parametri.

VARARGS

Dalla versione 1.5 di JAVA, esiste un meccanismo che si chiama varargs, che consente di definire dei metodi che hanno dei parametri formali indefiniti. Ciò vuol dire che, definendo un varargs, posso passare da 0 a n parametri.

Il varargs si definisce utilizzando il modificatore ... (3 punti).

Vediamo un esempio:

```
public class Varargs {  
  
    public String concatena(String... vars){  
        String out = "";  
  
        for(int i = 0; i < vars.length; i++){  
            out += vars[i];  
        }  
        return out;  
    }  
  
    public static void main(String[] args) {  
  
        Varargs a = new Varargs();  
        String d = a.concatena("Paolo", "Preite", "Corso", "Java");  
  
        System.out.println(d); //PaoloPreiteCorsoJava  
    }  
}
```

Per la JVM, definire un varargs equivale a definire un array, quindi posso iterare il varargs come se fosse un'array a tutti gli effetti.

MODIFICATORE DI ACCESSO AI METODI

I modificatori di accesso sono 4:

- **public**: il metodo è visibile a tutte le classi;
- **private**: il metodo è visibile solo alla classe che lo definisce;
- **protected**: il metodo è visibile solo alla classi che si trovano all'interno dello stesso package e dalle classi che estendono la classe che contiene il metodo;
- **default**: il metodo è visibile solo alle classi che si trovano nello stesso package nella classe in cui è stato definito il metodo.

Altri modificatori:

- **final**: utilizzato per rendere un metodo non ridefinibile (o non modificabile). Quindi, ciò non permette di fare l'override di questo metodo
- **static**: utilizzato per definire metodi associati ad una classe, ma non ad un'istanza. Questo vuol dire che i metodi statici non possono interagire con le variabili d'istanza, ma solamente con quelle statiche.

Invocazione di un metodo statico: `NomeClasse.nomeMetodo(...)`

Invocazione di un metodo non statico: `nomelistanza.nomeMetodo(...)`

Quindi, i metodi non statici sono associati ad ogni istanza di una classe, per cui il contesto di esecuzione è l'istanza stessa. I metodi non statici possono accedere alle variabili e ai metodi statici, ma il contrario non vale.

I MODIFICATORI (PIÙ IN GENERALE)

Il modificatore è una parola riservata al linguaggio JAVA.

Un modificatore va scritto prima della dichiarazione del componente, quindi prima di una classe, o prima di un metodo, o prima di una variabile. Ovviamente è possibile utilizzare anche più modificatori, oltre a quelli di accesso, senza un ordine preciso.

MODIFICATORI DI ACCESSO

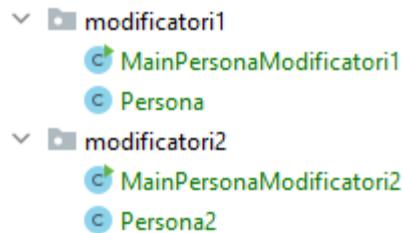
I modificatori di accesso, abbiamo visto, sono in grado di cambiare la visibilità e l'accesso ad un componente (classe, metodo, variabile).

Abbiamo visto anche che i modificatori di accesso sono 4:

- **public:** può essere utilizzato sia sulle classi, sia sui metodi e sia sulle variabili d'istanza (cioè le variabili definite all'interno di una classe, ma fuori dai metodi della classe stessa);
 - le classi public sono visibili da qualsiasi classe situata in qualsiasi package;
 - i metodi o le variabili di istanza public sono visibili da qualsiasi classe, in qualsiasi package;
- **protected:** può essere utilizzato sia sui metodi, sia sulle variabili d'istanza, ma non sulle classi, ed è leggermente più restrittivo del modificatore public;
 - un metodo o una variabile d'istanza protected è visibile da qualsiasi classe definita nello stesso package, o da tutte le classi che la ereditano, anche se si trovano in package diversi;
- **default (o senza modificatore):** può essere utilizzato sia sulle classi, sia sui metodi e sia sulle variabili d'istanza;
 - una classe definita senza modificatore è visibile solo dalle classi appartenenti dallo stesso package;
 - i metodi e le variabili d'istanza senza modificatore sono visibili solo alle classi che appartengono allo stesso package della classe che li definisce;
- **private:** può essere utilizzato sia sui metodi, sia sulle variabili d'istanza, ma non sulle classi, ed è il modificatore più restrittivo in assoluto;
 - un metodo o una variabile private è visibile solo all'interno della classe in cui è stato definito.

	Ovunque	Stessa classe	Stesso package	Classe derivata
public	SI	SI	SI	SI
protected	NO	SI	SI	SI
default	NO	SI	SI	NO
private	NO	SI	NO	NO

Per esempio:



Persona.java

```
package modificatori1;

public class Persona {
    private int id;
    private String nome;
    private String cognome;

    public void cammina() {
        /*...*/
    }

    protected void mangia() {
        /*...*/
    }

    void dormi(){
        /*...*/
    }

    private void bevi(){
        /*...*/
    }
}
```

MainPersonaModificatori1.java

```
package modificatori1;

public class MainPersonaModificatori1 {

    public static void main(String[] args) {
        Persona p = new Persona();

        p.cammina();
        p.dormi();
        p.bevi(); //bevi() has private access in modificatori1.Persona
        p.mangia();
    }
}
```

MainPersonaModificatori2.java

```
package modificatori2;

import modificatori1.Persona;

public class MainPersonaModificatori2 {

    public static void main(String[] args) {
        Persona p = new Persona();

        p.cammina();
        p.dormi(); //dormi() is not public in modificatori1.Persona;
        p.bevi(); //bevi() has private access in modificatori1.Persona
        p.mangia(); //mangia() has protected access in modificatori1.Persona
    }
}
```

Persona2.java

```
package modificatori2;

import modificatori1.Persona;

public class Persona2 extends Persona {
    @Override
    public void cammina() {
        super.cammina();
    }

    @Override
    protected void mangia() {
        super.mangia();
    }
}
```

ALTRI MODIFICATORI: FINAL E STATIC

Altri modificatori:

- **final:** può essere utilizzato sia sulle classi, sia sui metodi e sia sulle variabili. Questo modificatore indica che il componente non può essere modificato;
 - una classe final non può essere estesa;
 - non si può fare l'override di un metodo final;
 - un attributo final è una costante, cioè il suo valore non cambia mai. Per convenzione, tutte le variabili final hanno il nome in maiuscolo;
- **static:** può essere utilizzato sia sui metodi, sia sulle variabili, ma non sulle classi. Gli elementi static sono caricati in memoria insieme alla classe, ed appartengono alla classe e non all'oggetto;
 - un elemento static può essere usato anche se non esiste nessuna istanza;
 - un metodo static non può accedere ad elementi non static, poiché questi potrebbero non esistere in memoria

SINTASSI

CLASSI E INTERFACCE

La prima riga dei file contenenti le classi e le interfacce deve avere la definizione del package. Possiamo creare classi ed interfacce senza specificare il package, quindi utilizzando il package di default, però questa operazione è altamente sconsigliata.

SI

```
package it.corsi.java;  
public class NomeClasse {  
    ...  
}
```

NO (SCONSIGLIATO IL DEFAULT)

```
public class NomeClasse {  
    ...  
}
```

Prima del nome dell'elemento, è necessario specificare la sua visibilità e il tipo.

SI

```
package it.corsi.java;  
public interface NomeInterfaccia {  
    ...  
}
```

NO (SCONSIGLIATO IL DEFAULT)

```
NomeInterfaccia {  
    ...  
}
```

Dopo il nome dell'elemento, è possibile inserire altri elementi (ad esempio implements o extends)

SI

```
package it.corsi.java;  
public class NomeClasse extends ClasseA {  
    ...  
}
```

NO

```
public class NomeClasse extends {  
    ...  
}
```

Il contenuto della classe e dell'interfaccia è racchiuso tra parentesi graffe {}

SI

```
package it.corsi.java;  
public class NomeClasse {  
    private int id;  
}
```

NO

```
public class NomeClasse  
    private int id;
```

Una classe può contenere sia variabili, sia metodi. Se la classe è astratta, possiamo inserire all'interno della classe anche solo la definizione del metodo, senza l'implementazione.

```
package it.corsi.java;
public class NomeClasse {
    private String varA;

    public String getVarA() {
        return this.varA;
    }
}
```

Un'interfaccia può contenere variabili e solo le definizioni dei metodi, non le implementazioni.

SI

```
package it.corsi.java;
public interface NomeInterfaccia {
    public String getUser();
}
```

NO

```
package it.corsi.java;
public interface NomeInterfaccia {
    public String getUser();
    ...
}
```

METODI E VARIABILI

Prima del nome del metodo è necessario inserire la visibilità. Se non inseriamo la visibilità, viene considerata quella di default. È comunque sconsigliato usare quella di default, ma è meglio utilizzare sempre una visibilità che sia chiara.

Subito dopo la visibilità, è necessario specificare il tipo di dato ritornato. Se il metodo non ritorna niente, al posto del tipo di dato ritornato bisogna scrivere void.

SI

```
package it.corsi.java;
public class NomeClasse {

    public String calcolaCodiceFiscale(...) {
        ...
    }

    void stampaSomma() {
        ...
    }
}
```

NO

```
package it.corsi.java;
public class NomeClasse {

    calcolaCodiceFiscale(...) {
        ...
    }

    private stampaSomma() {
        ...
    }
}
```

Dopo il nome del metodo, è necessario specificare eventuali argomenti in input, racchiusi tra parentesi tonde (). Se un metodo ovviamente non ha parametri in ingresso, avremo semplicemente le parentesi tonde. Tutte le istruzioni che vengono eseguite all'interno di un metodo, sono racchiuse tra parentesi graffe {}.

SI

```
package it.corsi.java;
public class NomeClasse {
    public int calcolaSomma(int a, int b) {
        return a+b;
    }

    void stampaSomma() {
        System.out.println(calcolaSomma(3,4));
    }
}
```

NO

```
package it.corsi.java;
public class NomeClasse {
    public int calcolaSomma(a, b) {
        return a+b;
    }

    void stampaSomma()
        System.out.println(calcolaSomma(3,4));
    }
}
```

Un metodo può contenere al suo interno variabili ed una serie di istruzioni che consentono di eseguire un'operazione. Se le variabili sono variabili di classe o di istanza, quindi dichiarate al di fuori del metodo, bisogna definire la visibilità. In tutti i casi bisogna definire il tipo di dato che rappresenta questa variabile. La visibilità non deve essere inserita se le variabili sono variabili locali, quindi definite all'interno di metodi.

SI

```
package it.corsi.java;
public class NomeClasse {
    private int id;
    String nome;

    public int calcolaSomma(int a, int b) {
        return a+b;
    }

    void stampaSomma() {
        String out = calcolaSomma(3,4);
        System.out.println(out);
    }
}
```

NO

```
package it.corsi.java;
public class NomeClasse {
    private id;
    nome;

    public int calcolaSomma(int a, b) {
        return a+b;
    }

    void stampaSomma() {
        private String out = calcolaSomma(3,4);
        System.out.println(out);
    }
}
```

PACKAGE

Il package è uno strumento messo a disposizione da JAVA per raggruppare ed organizzare i vari elementi (classi, interfacce, enumeration, classi astratte e così via) all'interno del nostro software.

L'obiettivo principale del package è semplificare la lettura del codice sorgente e l'accesso a tali elementi, evitando eventuali situazioni di concorrenza tra classi.

JAVA stesso organizza i suoi elementi in package, in base alla loro funzione. Per esempio:

Esempi:

- **java.lang**: contiene le classi principali del linguaggio (Object, Class, System, String, StringBuffer,...)
- **java.util**: contiene le classi di utilità generica (Date, List, ArrayList, ...)
- **java.io**: contiene le classi per la gestione di input ed output (File, InputStream, OutputStream, FileInputStream, FileOutputStream)

Quando sviluppiamo un software, ovviamente anche noi dobbiamo creare i package in maniera tale da organizzare al meglio le nostre classi e i nostri elementi in generale.

La naming convention per i package prevede che essi siano scritti solo con lettere minuscole. Il package dovrebbe iniziare con l'estensione dei domini com, edu, gov, mil, net, org, oppure con le due lettere che identificano una nazione.

Ovviamente, avere package strutturati ci consente di gestire al meglio le nostre classi. Per esempio:

Esempio di organizzazione di package per una web application Java:

it.corsi.java.forms	conterrà classi di tipo “Form”
it.corsi.java.bean	conterrà classi di tipo “JavaBean”
it.corsi.java.actions	conterrà classi di tipo “Action”

COMANDO IMPORT

Import è una parola riservata a JAVA che consente di importare delle librerie esterne, anche librerie che non fanno parte del nostro package, permettendoci di utilizzarle nelle nostre classi.

Se vogliamo utilizzare una classe esterna al nostro progetto, innanzitutto dobbiamo specificare dove si trova il jar della libreria esterna che contiene le classi che vogliamo utilizzare, se la libreria non è disponibile tra quelle fornite da JAVA. Poi dobbiamo effettuare l'import, quindi importare le classi o il package (se vogliamo utilizzare più classi che si trovano all'interno dello stesso package).

Quindi, la dichiarazione di import ci consente di utilizzare le classi che sono presenti all'interno del package che stiamo importando.

L'import si effettua subito dopo la dichiarazione del package, prima della definizione della classe.

NAMING E CODE CONVENTION

Le naming e code convention sono delle indicazioni di tipo sintattico che consentono di rendere i sorgenti di un software facilmente leggibili anche da chi non l'ha scritto.

Non si tratta di regole obbligatorie, ma sono altamente consigliate.

Per quanto riguarda le naming e code convention dei package, li abbiamo visti nel paragrafo precedente.

NAMING E CODE CONVENTION PER LE CLASSI

Il nome della classe deve essere un sostantivo e deve iniziare con la lettera maiuscola.

Se il nome della classe contiene più sostantivi, ciascun sostantivo deve avere la prima lettera maiuscola.

È ammesso utilizzare classi che hanno tutte lettere maiuscole solamente per acronimi o abbreviazioni, come URL, HTML e così via.

Per le interfacce valgono le stesse regole delle classi.

NAMING E CODE CONVENTION PER LE VARIABILI

Il nome di una variabile deve indicare cosa rappresenta la variabile e deve iniziare con la lettera minuscola.

Se la variabile è composta da più parole, dalla seconda parola la prima lettera deve essere maiuscola.

È ammesso utilizzare nomi di variabili tutto in maiuscolo solo per le variabili di tipo costanti, che hanno il modificatore final. Se il nome di una variabile usata come costante è composta da più parole, si devono separare con il simbolo di underscore “_”.

Le variabili non devono mai iniziare con i simboli underscore “_” o dollaro “\$”.

Le singole lettere “i”, “j”, “k”... devono essere utilizzate solo come variabili temporanee (ad esempio in un ciclo for).

NAMING CONVENTION PER I METODI

I metodi devono essere verbi, oppure devono iniziare con un verbo.

Tutti i metodi devono iniziare con la lettera minuscola.

In caso di nome composto da più parole, anche qui a partire dalla seconda parola la prima lettera deve essere maiuscola.

COMANDI CONDIZIONALI

I comandi condizionali sono delle espressioni che consentono di eseguire una porzione di codice, se si verifica una determinata condizione che specifichiamo. Sostanzialmente funzionano in questo modo:

```
SE si verifica la condizione1  
    esegui queste istruzioni  
ALTRIMENTI SE si verifica la condizione2  
    esegui queste istruzioni  
...  
ALTRIMENTI  
    esegui queste istruzioni perché nessuna delle precedenti è soddisfatta
```

In JAVA abbiamo due tipi di comandi condizionali:

- if - else
- switch – case

IF – ELSE

Il comando if – else consente di eseguire una porzione di codice, solo se si verifica una condizione.

La sintassi è la seguente:

```
if(condizione_1) {  
    /* esegui queste istruzioni e non eseguire le successive */  
} else if(condizione_2) {  
    /* esegui queste istruzioni e non eseguire le successive */  
} else {  
    /* esegui queste istruzioni perché nessuna delle precedenti condizioni era vera */  
}
```

I comandi “else” ed “else if” non sono obbligatori.

Tutte le condizioni devono essere ovviamente di tipo boolean, quindi devono restituire o “true” o “false”.

Le parentesi graffe nell’if – else non sono obbligatorie, ma è caldamente consigliato di usarle sempre.

Vediamo un esempio:

```
public class IfElse {  
  
    public int recuperaIlMaggiore(int num1, int num2, int num3){  
        int maggiore = 0;  
  
        if(num1 > num2 && num1 > num3){  
            maggiore = num1;  
        } else if(num2 > num1 && num2 > num3){  
            maggiore = num2;  
        } else{  
            maggiore = num3;  
        }  
  
        return maggiore;  
    }  
}
```

```

public static void main(String[] args) {
    IfElse ie = new IfElse();
    int maggiore = ie.recuperaIlMaggiore(5, 3, 2);
    System.out.println(maggiorer); //5
}
}

```

SWITCH – CASE

Il comando switch -case è sempre un tipo di comando condizionale, che consente di eseguire una porzione di codice, anche qui, solo se si verifica una determinata condizione.

La sintassi è la seguente:

```

switch(parametro) {
    case valore_1:
        /* ...istruzioni... */
    case valore_2:
        /* ...istruzioni... */
    default:
        /* ...istruzioni... */
}

```

Logica di funzionamento: vengono eseguite le istruzioni a partire dal case che ha lo stesso valore di parametro.

Esempio: se parametro = valore_2 lo switch eseguirà tutte le istruzioni del case valore_2 e del case default

Il parametro passato in ingresso, a partire da JAVA 1.7, può essere anche String.

Uno switch si può interrompere attraverso il comando break.

Vediamo un esempio:

```

public class SwitchCase {

    public String switchSenzaBreak(int codice){
        String testo = null;

        switch (codice){
            case 1:
                testo = "codice 1";

            case 2:
                testo = "codice 2";
            case 3:
                testo = "codice 3";
        }
        return testo;
    }

    public String switchConBreak(int codice){
        String testo = null;

        switch (codice){
            case 1:
                testo = "codice 1";
                break;

            case 2:
                testo = "codice 2";
                break;
            case 3:
                testo = "codice 3";
        }
    }
}

```

```

        break;
    }
    return testo;
}

public static void main(String[] args) {

    SwitchCase sc = new SwitchCase();
    String testoDaSwitchSenzaBreak = sc.switchSenzaBreak(1);
    String testoDaSwitchConBreak = sc.switchConBreak(1);
    System.out.println(testoDaSwitchSenzaBreak); //codice 3
    System.out.println(testoDaSwitchConBreak); //codice 1

}
}

```

COMANDO RETURN

Questo comando è fondamentale perché, quando definiamo un metodo, consente di impostare il valore che vogliamo ritornare all'interno del metodo.

Il comando return può essere utilizzato solo se un metodo ha un parametro di ritorno specificato, quindi non può essere void. Tale comando può essere utilizzato anche più volte all'interno dello stesso metodo. Per esempio:

```

public class Return {

    public int recuperaIlMaggiore(int num1, int num2, int num3){
        if(num1 > num2 && num1 > num3){
            return num1;
        }else if(num2 > num1 && num2 > num3){
            return num2;
        }else{
            return num3;
        }
    }

    public String recuperaCodice(int codice){
        switch (codice){
            case 1:
                return "codice 1";
            case 2:
                return "codice 2";
            case 3:
                return "codice 3";
        }
        return "";
    }

    public static void main(String[] args) {
        Return r = new Return();
        int maggiore = r.recuperaIlMaggiore(5, 3, 2);
        String testo = r.recuperaCodice(1);
        System.out.println(maggiore); //5
        System.out.println(testo); //codice 1
    }
}

```

CICLI

I cicli consentono di eseguire ripetutamente un blocco di codice.

I cicli in JAVA sono 3:

- while
- do – while
- for

In tutte e 3 i cicli, le istruzioni sono racchiuse tra parentesi graffe {}. È possibile creare anche dei cicli senza inserire le parentesi graffe, però è caldamente consigliato utilizzarle sempre per semplificare la lettura del codice e per evitare eventuali bug applicativi inattesi.

WHILE

Il comando while esegue un blocco di codice finché non viene verificata la condizione specificata all'inizio del blocco di codice. La sintassi è la seguente:

```
while(condizione) {
    /* esegui queste istruzioni finché non si verifica la "condizione" */
}
```

Da fare attenzione al fatto che, se la condizione del while è sempre vera, una volta che il software entra all'interno del while, andrà in loop infinito e non terminerà mai.

DO – WHILE

Il comando do – while è simile al comando while, solo che nel do – while la condizione viene verificata alla fine del blocco di codice e non all'inizio.

Nel do – while l'esecuzione del blocco di codice viene effettuata almeno una volta, anche se la condizione è falsa. Anche in questo caso è importante che la condizione diventi falsa, altrimenti il ciclo andrà in loop infinito. La sintassi è la seguente:

```
do {
    /* esegui queste istruzioni finché non si verifica la "condizione" */
} while(condizione);
```

FOR

Il comando for esegue un blocco di codice finché non viene verificata la condizione specificata. La sintassi è la seguente:

```
for(inizializzazione; condizione; incremento) {
    /* esegui queste istruzioni finché si verifica la "condizione" */
}
```

- l'inizializzazione è un'espressione che contiene la definizione e l'inizializzazione di una variabile. Consente di definire una variabile che utiliziamo come contatore;
- la condizione è l'espressione che viene verificata;
- l'incremento è un'espressione che consente di aumentare o diminuire la variabile (che abbiamo definito in fase di inizializzazione) ad ogni iterazione.

Anche qui posso creare un loop infinito, se non specifico alcun parametro.

ESEMPIO DI TUTTI E 3 I CICLI

Per esempio:

```
public class ForWhileDoWhile {

    public void iteraConWhile(int contatore, int estremo){
        while(contatore < estremo){
            System.out.println("contatore vale" + contatore);
            contatore++;
        }
        System.out.println("sono uscito dal ciclo while\n");
    }

    public void iteraConDoWhile(int contatore, int estremo){
        do{
            System.out.println("contatore vale" + contatore);
            contatore++;
        }while(contatore < estremo);

        System.out.println("sono uscito dal ciclo do-while\n");
    }

    public void iteraConFor(int contatore){
        for(int i = 0; i < contatore; i++){
            System.out.println(i);
        }
        System.out.println("sono uscito dal ciclo for\n");
    }

    public static void main(String[] args) {
        ForWhileDoWhile fwdw = new ForWhileDoWhile();
        fwdw.iteraConWhile(0,5);
        fwdw.iteraConWhile(5,5);
        fwdw.iteraConDoWhile(0,5);
        fwdw.iteraConDoWhile(5,5);
        fwdw.iteraConFor(5);
    }
}
```

L'output di tale codice è il seguente:

```
contatore vale0  
contatore vale1  
contatore vale2  
contatore vale3  
contatore vale4  
sono uscito dal ciclo while
```

```
sono uscito dal ciclo while
```

```
contatore vale0  
contatore vale1  
contatore vale2  
contatore vale3  
contatore vale4  
sono uscito dal ciclo do-while
```

```
contatore vale5  
sono uscito dal ciclo do-while
```

```
0  
1  
2  
3  
4  
sono uscito dal ciclo for
```

COMANDI DI INTERRUZIONE DI CICLO

I comandi di interruzione di ciclo sono delle parole riservate a JAVA, che ci consentono di interrompere i cicli. I comandi di interruzione di ciclo sono 2:

- break;
- continue.

BREAK

Il comando break è utilizzato per terminare l'esecuzione di un blocco di codice in un ciclo o uno switch. Quando il nostro software, durante l'esecuzione, incontra l'istruzione break, esce da un' iterazione e la interrompe, non eseguendo neanche le successive.

CONTINUE

Il comando continue è utilizzato per terminare l'iterazione corrente, dicendo al software di proseguire con l'iterazione successiva.

ESEMPIO CON BREAK E CONTINUE

```
public class BreakContinue {  
  
    public void Break(int interruttore, int estremo){  
  
        for(int i = 0; i < estremo; i++){  
            if(i == interruttore){  
                System.out.println("i == interruttore!!!");  
                break;  
            }  
            System.out.println("Iterazione numero " + i);  
        }  
  
        System.out.println("ciclo con break\n");  
    }  
  
    public void Continue(int interruttore, int estremo){  
  
        for(int i = 0; i < estremo; i++){  
            if(i == interruttore){  
                System.out.println("i == interruttore!!!");  
                continue;  
            }  
            System.out.println("Iterazione numero " + i);  
        }  
  
        System.out.println("ciclo con continue\n");  
    }  
  
    public static void main(String[] args) {  
  
        BreakContinue bc = new BreakContinue();  
        bc.Break(5, 10);  
        bc.Continue(5,10);  
    }  
}
```

L'output di tale codice è il seguente:

```
Iterazione numero 0  
Iterazione numero 1  
Iterazione numero 2  
Iterazione numero 3  
Iterazione numero 4  
i == interruttore!!!  
ciclo con break
```

```
Iterazione numero 0  
Iterazione numero 1  
Iterazione numero 2  
Iterazione numero 3  
Iterazione numero 4  
i == interruttore!!!  
Iterazione numero 6  
Iterazione numero 7  
Iterazione numero 8  
Iterazione numero 9  
ciclo con continue
```

PROGRAMMAZIONE ORIENTATA AGLI OGGETTI

La programmazione orientata agli oggetti è un approccio alla programmazione nato tra la seconda metà degli anni 50 e l'inizio degli anni 60.

La caratteristica fondamentale di questo paradigma è che consente di rappresentare un problema, o delle entità reali, attraverso oggetti software e consente di stabilire delle relazioni che intercorrono tra queste entità reali.

Nella programmazione funzionale, lo sviluppo di un programma è basato su variabili e funzioni (o metodi). Nella programmazione a oggetti, invece, lo sviluppo è basato su entità chiamate classi che contengono al loro interno variabili (attributi della classe) e funzioni (metodi che definiscono il comportamento).

Per esempio, pensando ad uno smartphone, nella programmazione funzionale avremmo una serie di array che rappresentano ogni singolo componente, intersecati tra di loro attraverso delle funzioni. Invece, nella programmazione a oggetti avremmo una serie di classi che rappresentano ogni singolo componente, al cui interno mettere variabili e funzioni.

Tra i vantaggi della programmazione orientata agli oggetti abbiamo:

- il supporto alla modellazione software di oggetti del mondo reale e del mondo astratto da riprodurre;
- una migliore gestione e manutenzione dei software di grandi dimensioni;
- favorisce lo sviluppo modulare di software ed il riuso del codice.

I concetti base della programmazione orientata agli oggetti sono:

- classe, che rappresenta un tipo di dato;
- attributo, che rappresenta la proprietà di una classe ed è definita attraverso la variabile;
- metodo, che rappresenta l'azione che una classe può eseguire;
- oggetto o istanza, che è la rappresentazione fisica di una classe.

Vediamo un esempio pratico, modellando la classe smartphone:

Smartphone.java

```
public class Smartphone {

    private String serialNumber;
    private String imei;
    private String marca;
    private String modello;
    private Display schermo;

    public String getSerialNumber() {
        return serialNumber;
    }

    public void setSerialNumber(String serialNumber) {
        this.serialNumber = serialNumber;
    }
}
```

```

public String getImei() {
    return imei;
}

public void setImei(String imei) {
    this.imei = imei;
}

public String getMarca() {
    return marca;
}

public void setMarca(String marca) {
    this.marca = marca;
}

public String getModello() {
    return modello;
}

public void setModello(String modello) {
    this.modello = modello;
}
}

```

Display.java

```

public class Display {

    private String marca;
    private String modello;
    private String risoluzione;

    public String getMarca() {
        return marca;
    }

    public void setMarca(String marca) {
        this.marca = marca;
    }

    public String getModello() {
        return modello;
    }

    public void setModello(String modello) {
        this.modello = modello;
    }

    public String getRisoluzione() {
        return risoluzione;
    }

    public void setRisoluzione(String risoluzione) {
        this.risoluzione = risoluzione;
    }
}

```

MainSmartphone.java

```
public class MainSmartphone {  
  
    public static void main(String[] args) {  
        Smartphone sm1 = new Smartphone();  
        sm1.setImei("123123");  
        sm1.setMarca("Apple");  
        sm1.setModello("iPhone 7");  
  
        Smartphone sm2 = new Smartphone();  
        sm2.setImei("6576575");  
        sm2.setMarca("Samsung");  
        sm2.setModello("Note7");  
    }  
}
```

INCAPSULAMENTO (parte 1)

L'incapsulamento è una tecnica che consente di nascondere il funzionamento interno di una porzione di programma, quindi permette di dichiarare cosa si fa, ma non come. Per la programmazione a oggetti, l'incapsulamento si traduce nel fatto di nascondere gli attributi, quindi rendere gli attributi visibili solo all'interno della classe con la keyword private e consentire l'accesso gli attributi attraverso dei metodi getter e setter con visibilità public.

L'incapsulamento si basa sul principio dell'information hiding, cioè sapere cosa fa una classe, ma non come lo fa. Detto ciò, possiamo inserire all'interno dei metodi getter e setter delle operazioni che non vogliamo far vedere all'utente. Vediamo un esempio con i metodi getter e setter di imei della classe Smartphone:

```
public String getImei() {  
    String tmp = imei.concat("---");  
    return tmp;  
}  
  
public void setImei(String imeitmp) {  
    if(imeitmp.length() < 20){  
        imeitmp.concat("sos94");  
    }  
  
    this.imei = imeitmp;  
}
```

Le modifiche all'interno dei getter e setter vengono considerate solo se la variabile a cui si riferiscono è di tipo private, altrimenti vengono ignorate dal compilatore.

EREDITARIETÀ (parte 1)

L'ereditarietà è una tecnica che consente di estendere delle caratteristiche di una classe ad un'altra classe.

Una classe che eredita da un'altra classe ha le seguenti peculiarità:

- mantiene metodi ed attributi della classe da cui deriva;
- può definire i propri metodi o attributi;
- può ridefinire il codice di alcuni dei metodi ereditati mediante un meccanismo chiamato overriding.

Esempio	Ereditata da Prodotto	Ereditata da Prodotto																							
<table border="1"><tr><td>Prodotto</td></tr><tr><td><i>Attributi</i></td></tr><tr><td>- id</td></tr><tr><td>- nome</td></tr><tr><td>- descrizione</td></tr><tr><td>..</td></tr><tr><td><i>Metodi</i></td></tr><tr><td>- List getStores()</td></tr><tr><td>...</td></tr></table>	Prodotto	<i>Attributi</i>	- id	- nome	- descrizione	..	<i>Metodi</i>	- List getStores()	...	<table border="1"><tr><td>Smartphone</td></tr><tr><td><i>Stessi attributi di Prodotto ed inoltre</i></td></tr><tr><td>- dimensioni</td></tr><tr><td>- memoria</td></tr><tr><td>- processore</td></tr><tr><td>..</td></tr><tr><td><i>Stessi metodi di Prodotto e possibilità di crearne di nuovi...</i></td></tr></table>	Smartphone	<i>Stessi attributi di Prodotto ed inoltre</i>	- dimensioni	- memoria	- processore	..	<i>Stessi metodi di Prodotto e possibilità di crearne di nuovi...</i>	<table border="1"><tr><td>Libro</td></tr><tr><td><i>Stessi attributi di Prodotto ed inoltre</i></td></tr><tr><td>- numeroPagine</td></tr><tr><td>- autore</td></tr><tr><td>- editore</td></tr><tr><td>..</td></tr><tr><td><i>Stessi metodi di Prodotto e possibilità di crearne di nuovi...</i></td></tr></table>	Libro	<i>Stessi attributi di Prodotto ed inoltre</i>	- numeroPagine	- autore	- editore	..	<i>Stessi metodi di Prodotto e possibilità di crearne di nuovi...</i>
Prodotto																									
<i>Attributi</i>																									
- id																									
- nome																									
- descrizione																									
..																									
<i>Metodi</i>																									
- List getStores()																									
...																									
Smartphone																									
<i>Stessi attributi di Prodotto ed inoltre</i>																									
- dimensioni																									
- memoria																									
- processore																									
..																									
<i>Stessi metodi di Prodotto e possibilità di crearne di nuovi...</i>																									
Libro																									
<i>Stessi attributi di Prodotto ed inoltre</i>																									
- numeroPagine																									
- autore																									
- editore																									
..																									
<i>Stessi metodi di Prodotto e possibilità di crearne di nuovi...</i>																									

POLIMORFISMO (parte 1)

Il polimorfismo è una tecnica che consente di sovrascrivere delle caratteristiche o azioni di un metodo. È una caratteristica strettamente legata all'ereditarietà, quindi se una classe eredita un'altra classe, la nuova classe potrà ridefinire le azioni che effettua la classe madre.

Il polimorfismo si effettua con l'overriding e consiste nel realizzare in ciascuna classe figlia un'implementazione del metodo della classe madre.

Esempio	Ereditata da Prodotto	Ereditata da Prodotto						
<table border="1"><tr><td>Prodotto</td></tr><tr><td>- List getStores()</td></tr></table>	Prodotto	- List getStores()	<table border="1"><tr><td>Smartphone</td></tr><tr><td>Sovrascrivo il metodo getStores() recuperando la lista dei negozi dov'è possibile acquistare il libro da una tabella dei negozi di ecommerce</td></tr></table>	Smartphone	Sovrascrivo il metodo getStores() recuperando la lista dei negozi dov'è possibile acquistare il libro da una tabella dei negozi di ecommerce	<table border="1"><tr><td>Libro</td></tr><tr><td>Sovrascrivo il metodo getStores() recuperando la lista dei negozi dov'è possibile acquistare il libro da una tabella dei negozi fisici</td></tr></table>	Libro	Sovrascrivo il metodo getStores() recuperando la lista dei negozi dov'è possibile acquistare il libro da una tabella dei negozi fisici
Prodotto								
- List getStores()								
Smartphone								
Sovrascrivo il metodo getStores() recuperando la lista dei negozi dov'è possibile acquistare il libro da una tabella dei negozi di ecommerce								
Libro								
Sovrascrivo il metodo getStores() recuperando la lista dei negozi dov'è possibile acquistare il libro da una tabella dei negozi fisici								

CONSEGUENZE A LIVELLO HARDWARE DOPO AVER ISTANZIATO UNA CLASSE

Nel nostro dispositivo, ovviamente, abbiamo il processore che effettua i calcoli, l'hard disk su cui vengono salvate le informazioni e la RAM.

Nel caso specifico della programmazione a oggetti, ogni istanza di una classe viene associata ad una particolare area della memoria.

Creare un'istanza, quindi, vuol dire riservare una porzione di memoria, o allocare memoria.

Se la classe viene inizializzata, la porzione viene anche riempita con i valori passati in fase di inizializzazione

JAVA consente di inizializzare automaticamente un oggetto attraverso dei metodi chiamati costruttori. Il costruttore consente di creare un'istanza di una classe.

Quando un oggetto non viene più utilizzato, la memoria occupata da quell'oggetto si può eliminare attraverso un metodo chiamato distruttore.

JAVA non consente di eliminare gli oggetti dalla memoria, ma ha un meccanismo, chiamato garbage collector, che in automatico libera la memoria quando gli oggetti non sono più utilizzati.

Esempio

```
private String produttore;
```

Creo un'istanza della classe **String** e la chiamo **produttore**. Il nostro dispositivo riserva una porzione di memoria alla nostra istanza produttore.

```
produttore = new String("Utilizzo il costruttore");
```

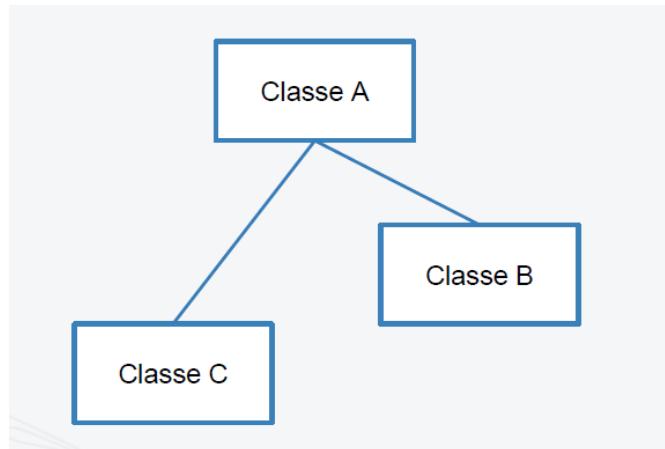
Inizializzo la variabile produttore attraverso il costruttore della classe String.

In questa fase, il nostro dispositivo riempirà lo spazio di memoria riservato all'istanza con il valore assegnato (cioè la frase "Utilizzo il costruttore")

RELAZIONE TRA CLASSI

Nella programmazione a oggetti, molto spesso una classe ha la necessità di interraccarsi con una o più classi per scambiare informazioni, per scambiare messaggi o per invocare l'esecuzione di un metodo.

Il rapporto che esiste tra due o più classi consente di determinare e di identificare il tipo di relazione che intercorre tra esse.



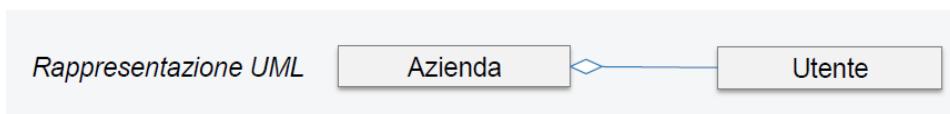
Un esempio classico è considerare la classe Prodotto e la classe CategoriaProdotto. Una classe Prodotto può appartenere a una o più CategoriaProdotto.

Le relazioni tra classi sono:

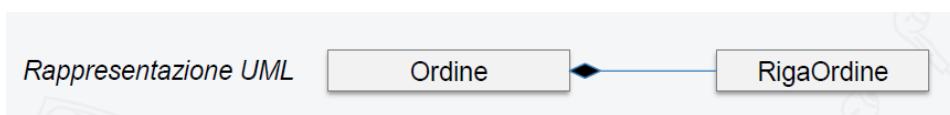
- Associazione



- Aggregazione



- Composizione

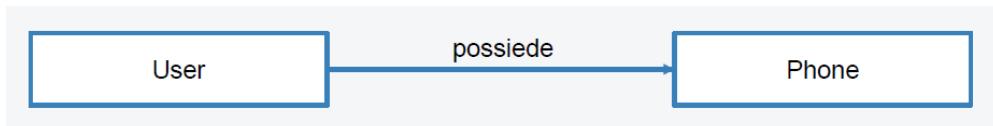


ASSOCIAZIONE

L'associazione è la relazione tra due classi più diffusa.

Si ha una relazione di associazione tra la classe A e la classe B se, da un'istanza della classe A, è possibile accedere ad un'istanza della classe B, o viceversa.

Ad esempio, un utente può avere più numeri di telefono.



Dal punto di vista della modellazione UML, è possibile assegnare un nome (ad es <<possiede>>) e una direzione all'associazione. La direzione serve ad indicare il verso in cui avviene la navigazione tra le classi, cioè a partire da quale classe posso accedere alle informazioni dell'altra classe.

Se la navigazione è bidirezionale, non è necessario inserire delle frecce.

Esempio <pre>graph LR; Ordine[Ordine] -- Effettuato da --> Utente[Utente]</pre> <p>The diagram shows two rectangular boxes: 'Ordine' on the left and 'Utente' on the right. A blue arrow points from 'Ordine' to 'Utente', labeled 'Effettuato da' above the arrowhead. This represents a directed association where an order is performed by a user.</p>	<p>La relazione indica che un ordine di acquisto è effettuato da un utente. A partire dall'ordine è possibile risalire all'utente che l'ha creato. Il contrario non si può fare.</p>
Esempio <pre>graph LR; Utente[Utente] -- Possiede --> Ruolo[Ruolo]</pre> <p>The diagram shows two rectangular boxes: 'Utente' on the left and 'Ruolo' on the right. A blue arrow points from 'Utente' to 'Ruolo', labeled 'Possiede' above the arrowhead. This represents a directed association where a user has roles.</p>	<p>La relazione indica che un utente ha uno o più ruoli. A partire dall'utente è possibile risalire ai ruoli associati. Il contrario non si può fare.</p>
Esempio <pre>graph LR; ContoCorrente[ContoCorrente] -- Appartiene a --> Utente[Utente]</pre> <p>The diagram shows two rectangular boxes: 'ContoCorrente' on the left and 'Utente' on the right. A blue arrow points from 'ContoCorrente' to 'Utente', labeled 'Appartiene a' above the arrowhead. This represents a directed association where a current account belongs to a user.</p>	<p>La relazione indica che un conto corrente appartiene ad uno o più utenti. La relazione è bidirezionale poiché a partire dall'utente è possibile risalire al suo conto corrente e viceversa.</p>

ECCEZIONI

Le eccezioni sono delle situazioni anomale e inaspettate, che si possono verificare durante l'esecuzione di un programma.

Quando si verifica delle eccezioni, a seconda della loro gravità, possiamo avere l'interruzione del programma, generare output inattesi, inserire delle righe sul database che non andavano inserite, effettuare il tracciamento di attività inattese e così via. Le situazioni che possono generare le eccezioni sono molteplici.

Quando si sviluppa un nuovo software, è importante prevenire l'errore e, ancor di più, gestire eventuali situazioni anomale.

Per esempio, supponiamo di avere un metodo che calcola la divisione tra due numeri. Se il divisore è 0, avremo un'eccezione che può, se non è gestita correttamente, interrompere l'esecuzione del software.

Un altro esempio ancor più frequente ce lo abbiamo quando, nella dichiarazione delle variabili, non è presente alcuna inizializzazione.

Le eccezioni possono essere di due tipi:

- eccezioni controllate (checked)
- eccezioni non controllate (unchecked)

ECCEZIONI CHECKED

Le eccezioni checked sono dovute a eventi che si verificano esternamente al nostro software.

L'esempio classico è durante l'utilizzo delle classi File, FileReader e FileWriter per leggere un file di configurazione, o di testo. Se il file non esiste, avremo un'eccezione.

Questi tipi di eccezioni si chiamano controllate perché, già in fase di compilazione, il compilatore ci segnala la mancata gestione di queste eccezioni. Vediamo un esempio in JAVA:

```
import java.io.File;
import java.io.FileReader;

public class MainEccezioni {

    public static void main(String[] args){

        File f = new File("test.txt");

        f.exists();

        FileReader fr = new FileReader(f); //exception java.io.FileNotFoundException
        /*se il file non esiste*/
    }
}
```

ECCEZIONI UNCHECKED

Le eccezioni non controllate sono i famosi bug, non vengono rilevate dal compilatore, però si possono verificare durante l'esecuzione di un programma. Vediamo un esempio in JAVA:

```
public class EccezioniUnchecked {  
  
    public static void main(String[] args) {  
  
        EccezioniUnchecked eu = new EccezioniUnchecked();  
        eu.stampaTesto(null);  
        //at eccezione.EccezioniUnchecked.main(EccezioniUnchecked.java:8)  
  
        eu.stampaTesto("testo di prova");/*non viene eseguito perché è stata Lanciata  
un'eccezione*/  
  
    }  
  
    private void stampaTesto(String testo){  
  
        String tmp = testo.concat("...");/*punto in cui è Lanciata l'eccezione*/  
        //at eccezione.EccezioniUnchecked.stampaTesto(EccezioniUnchecked.java:16)  
  
        System.out.println(tmp);/*non viene eseguito perché è stata Lanciata  
un'eccezione*/  
    }  
}
```

MECCANISMI DI GESTIONE DELLE ECCEZIONI

JAVA consente di gestire in maniera molto elegante le eccezioni che si possono verificare all'interno del nostro software, in quanto le eccezioni sono oggetti di classi particolari.

Quando in un metodo si verifica un'eccezione, tale metodo la lancia e la passa al metodo chiamante.

Ritorniamo all'esempio precedente per capire questo concetto:

The screenshot shows an IDE interface with two main parts. The top part is a code editor displaying Java code for a class named 'EccezioniUnchecked'. The code contains a main method that creates an instance of EccezioniUnchecked and calls its stampaTesto method with a null parameter. The stampaTesto method concatenates a string with null and prints it to the console. The bottom part is a terminal window titled 'EccezioniUnchecked' showing the execution of the Java program. It outputs the path to the Java executable, the exception stack trace ('Exception in thread "main" java.lang.NullPointerException'), and the exit code (1).

```
1 package eccezione;
2
3 public class EccezioniUnchecked {
4
5     public static void main(String[] args) {
6
7         EccezioniUnchecked eu = new EccezioniUnchecked();
8         eu.stampaTesto(null);
9         //at eccezione.EccezioniUnchecked.main(EccezioniUnchecked.java:8)
10
11        eu.stampaTesto("testo di prova");/*non viene eseguito perché è stata lanciata un'eccezione*/
12    }
13
14    @ private void stampaTesto(String testo){
15
16        String tmp = testo.concat("...");/*punto in cui è lanciata l'eccezione*/
17        //at eccezione.EccezioniUnchecked.stampaTesto(EccezioniUnchecked.java:16)
18
19        System.out.println(tmp);/*non viene eseguito perché è stata lanciata un'eccezione*/
20    }
21
22 }
23
```

EccezioniUnchecked x
C:\Users\spanico\.jdks\corretto-1.8.0_312\bin\java.exe ...
Exception in thread "main" java.lang.NullPointerException Create breakpoint
at eccezione.EccezioniUnchecked.stampaTesto(EccezioniUnchecked.java:16)
at eccezione.EccezioniUnchecked.main(EccezioniUnchecked.java:8)
Process finished with exit code 1

Nell'esempio abbiamo il metodo stampaTesto(), che prende come parametro in ingresso la stringa testo, utilizza questo parametro per fare una concatenazione e, infine, stampa il valore della variabile tmp, che contiene il risultato della concatenazione. Se passiamo null al metodo, abbiamo già visto che abbiamo un'eccezione NullPointerException. Tale eccezione è stata lanciata dal metodo stampaTesto() ed è stata passata al metodo chiamante, che in questo caso è il main(). Quindi, se leggiamo la console dopo la compilazione del sorgente, la prima riga dell' "at" indica il metodo in cui è stata generata l'eccezione (ossia il metodo che ha lanciato l'eccezione), dopodiché a cascata ci sono tutti i metodi che sono stati invocati prima di arrivare al metodo che ha generato l'eccezione.

Il metodo nel quale si verifica l'eccezione termina l'esecuzione, senza eseguire il resto delle istruzioni nel corpo del metodo.

Quando l'eccezione raggiunge il metodo main(), l'esecuzione del programma termina e viene stampato il messaggio di errore.

Se inseriamo qualche piccola modifica all'esempio precedente, possiamo notare che comunque il compilatore ci mostrerà lo stesso l'eccezione e tutti i metodi invocati a cascata, seguendo lo stesso principio di prima:

```

1 package eccezione;
2
3 public class EccezioniUnchecked {
4
5     public static void main(String[] args) {
6
7         EccezioniUnchecked eu = new EccezioniUnchecked();
8         eu.stampaTesto2 (null);
9         //at eccezione.EccezioniUnchecked.main(EccezioniUnchecked.java:8)
10
11         eu.stampaTesto("testo di prova");/*non viene eseguito perché è stata lanciata un'eccezione*/
12     }
13
14     @
15     private void stampaTesto(String testo){
16
17         String tmp = testo.concat("...");/*punto in cui è lanciata l'eccezione*/
18         //at eccezione.EccezioniUnchecked.stampaTesto(EccezioniUnchecked.java:16)
19
20         System.out.println(tmp);/*non viene eseguito perché è stata lanciata un'eccezione*/
21     }
22
23     private void stampaTesto2(String testo){
24
25         stampaTesto(testo);
26     }
27 }
```

EccezioniUnchecked

C:\Users\spanico\.jdks\corretto-1.8.0_312\bin\java.exe ...
Exception in thread "main" java.lang.NullPointerException Create breakpoint
at eccezione.EccezioniUnchecked.stampaTesto(EccezioniUnchecked.java:17)
at eccezione.EccezioniUnchecked.stampaTesto2(EccezioniUnchecked.java:25)
at eccezione.EccezioniUnchecked.main(EccezioniUnchecked.java:8)

Process finished with exit code 1

Proviamo ora a gestire l'eccezione nel metodo stampaTesto2() per vedere quale sarà il nuovo output:

```

public class EccezioniUnchecked {

    public static void main(String[] args) {

        EccezioniUnchecked eu = new EccezioniUnchecked();
        eu.stampaTesto2 (null);

        eu.stampaTesto("testo di prova");

    }

    private void stampaTesto(String testo){

        String tmp = testo.concat("...");

        System.out.println(tmp); // testo di prova...
    }
}
```

```

private void stampaTesto2(String testo){

    try{
        stampaTesto(testo);
    }catch(Exception e) {}
}

}

```

In questo caso è successo che è stato chiamato il metodo stampaTesto2(), che a sua volta chiama il metodo stampaTesto(). Il metodo stampaTesto() lancia l'eccezione che viene, però, intercettata nel metodo stampaTesto2() nel blocco try/catch, per cui la chiamata al metodo stampaTesto2() nel main() va a buon fine. Infine, il software esegue come se nulla fosse tutto il resto delle istruzioni.

Detto ciò, per quanto riguarda l'intercettare e gestire le eccezioni, possiamo fare 2 cose:

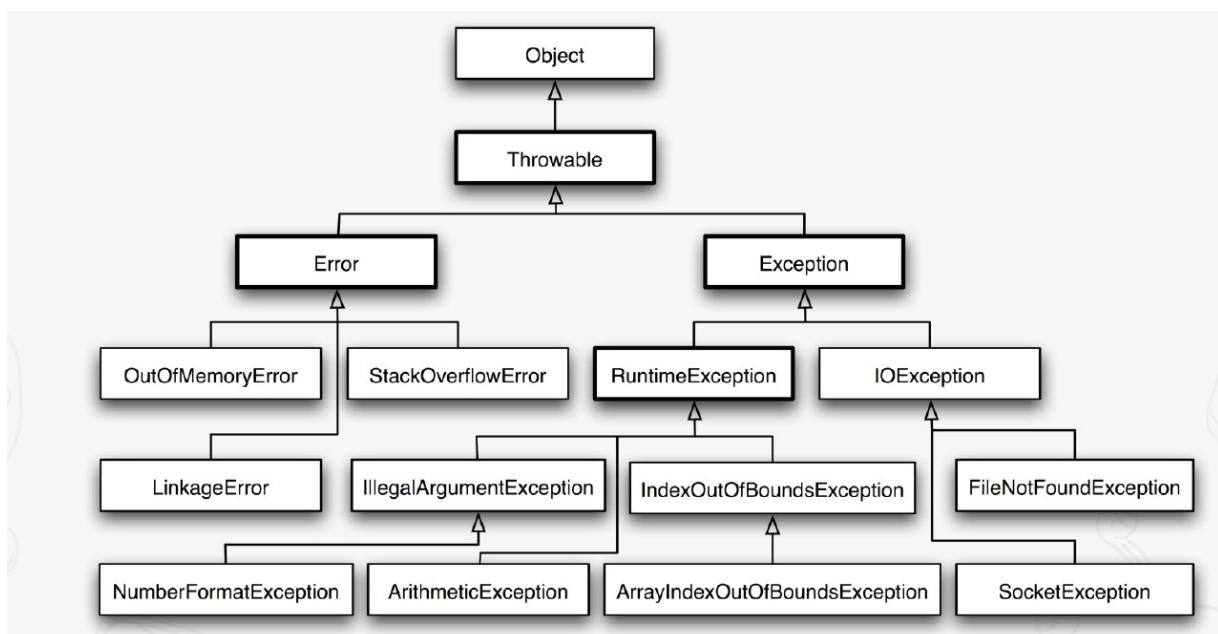
- catturare l'eccezione e decidere come gestirla;
- proseguire l'esecuzione del programma, ignorando l'eccezione (come nell'esempio precedente)

Le eccezioni in JAVA, abbiamo detto, sono istanze di sottoclassi della classe java.lang.Throwable.

Utilizzando il principio dell'ereditarietà, è possibile definire gerarchie di eccezioni. Tali gerarchie servono per identificare in maniera più rapida (ed elegante) le varie eccezioni, consentendo di gestirle nel modo opportuno.

In JAVA esiste una gerarchia di eccezioni predefinite, tuttavia siamo liberi di definire nuove sottoclassi di eccezioni in base alle nostre esigenze.

La classe Throwable, che è la classe principale per quanto riguarda le eccezioni, ha due sottoclassi: java.lang.Error e java.lang.Exception.



Le sottoclassi di Error vengono invocate quando si verificano dei problemi irrecuperabili relativi al funzionamento della JVM (memoria esaurita, crash del disco, ...), perciò non vanno né catturati né gestiti.

Le sottoclassi di Exception vengono invocate quando si verificano situazioni potenzialmente recuperabili (indice di un array oltre i limiti, file da leggere o su cui scrivere non trovato, formato di un numero errato,...), che vanno intercettate e gestite.

Ora vediamo un esempio di creazione di un'eccezione personalizzata:

Prima di scrivere l'esempio, però, bisogna sapere che per creare un'eccezione personalizzata, è necessario definire una classe che eredita le proprietà della classe java.lang.RuntimeException.

CorsoJavaException.java

```
import java.io.PrintStream;
import java.io.PrintWriter;

public class CorsoJavaException extends Throwable{

    /*Il serialVersionUID si genera da IDE*/
    private static final long serialVersionUID = 181090653271288169L;

    /*Fare override di tutti i metodi della classe Throwable*/
    /*Per il nostro esempio, effettuiamo modifiche nel metodo
     * getMessage()*/
    
    public CorsoJavaException() {
        super();
    }

    public CorsoJavaException(String message) {
        super(message);
    }

    public CorsoJavaException(String message, Throwable cause) {
        super(message, cause);
    }

    public CorsoJavaException(Throwable cause) {
        super(cause);
    }

    protected CorsoJavaException(String message, Throwable cause, boolean
enableSuppression, boolean writableStackTrace) {
        super(message, cause, enableSuppression, writableStackTrace);
    }

    @Override
    public String getMessage() {
        /*return super.getMessage();*/
        return "ATTENZIONE: nel progetto Corso_JAVA_EE si e' verificato un errore!!!";
    }

    @Override
    public String getLocalizedMessage() {
        return super.getLocalizedMessage();
    }

    @Override
```

```

public synchronized Throwable getCause() {
    return super.getCause();
}

@Override
public synchronized Throwable initCause(Throwable cause) {
    return super.initCause(cause);
}

@Override
public String toString() {
    return super.toString();
}

@Override
public void printStackTrace() {
    super.printStackTrace();
}

@Override
public void printStackTrace(PrintStream s) {
    super.printStackTrace(s);
}

@Override
public void printStackTrace(PrintWriter s) {
    super.printStackTrace(s);
}

@Override
public synchronized Throwable fillInStackTrace() {
    return super.fillInStackTrace();
}

@Override
public StackTraceElement[] getStackTrace() {
    return super.getStackTrace();
}

@Override
public void setStackTrace(StackTraceElement[] stackTrace) {
    super.setStackTrace(stackTrace);
}
}

```

MainEccezionePersonalizzata.java

```

public class MainEccezionePersonalizzata {

    public static void main(String[] args) {
        MainEccezionePersonalizzata mainEccezionePersonalizzata = new
MainEccezionePersonalizzata();
        try {
            mainEccezionePersonalizzata.stampaTesto(null);
        } catch (CorsoJavaException e) {
            /*e.printStackTrace();*/
            System.out.println(e.getMessage());
        }
    }
}

```

```

    }

    public void stampaTesto(String testo) throws CorsoJavaException {
        if(testo == null){
            throw new CorsoJavaException();
        }
    }
}

```

TRY – CATCH – FINALLY

Il blocco try - catch - finally consente di eseguire un programma, intercettando e gestendo l'eventuale eccezione che si verifica.

Sintassi

```

try {
    /* blocco di istruzioni da eseguire */
} catch {
    /* istruzioni da eseguire quando viene intercettata un'eccezione */
} finally {
    /* istruzioni da eseguire al termine del blocco di istruzioni try o catch */
}

```

Vediamo alcune regole da seguire per utilizzare il blocco try – catch – finally:

- è possibile avere più blocchi catch per gestire diversi tipi di eccezioni;
- è possibile avere solo un blocco try ed un blocco finally;
- se è presente almeno un blocco catch, il blocco finally può essere omesso.

Il blocco try – catch – finally ha il seguente comportamento:

- se le istruzioni che si trovano all'interno del blocco try non generano eccezioni ed è presente il blocco finally, vengono eseguite le istruzioni che si trovano nel finally. In questo caso, tutto quello che si trova nel blocco catch non verrà eseguito;
- se un'istruzione presente all'interno del blocco try genera un'eccezione, viene interrotta l'esecuzione delle istruzioni che seguono il punto in cui si è verificata l'eccezione, vengono eseguite le istruzioni che si trovano nel catch e, in seguito, quelle che si trovano nel finally, se presente;
- le istruzioni presenti nel blocco finally vengono eseguite sempre, anche in caso di eccezioni, se presente.

L'ultima regola è molto importante nel caso in cui, per esempio, si sta lavorando con file e database ed è necessario che il blocco di istruzioni che chiude il file o la connessione al database sia sempre eseguito ad un certo punto del codice.

THROW E THROWS

Il comando throw viene utilizzato per lanciare un'eccezione durante l'esecuzione del software.

Tale comando consiste semplicemente nello scrivere la keyword throw, seguita da new Exception(), ossia l'oggetto della classe Exception che estende Throwable. Dopodichè si può passare eventualmente a Exception un messaggio come parametro in ingresso e, quando la JVM in fase di esecuzione intercetta la riga in cui è presente la keyword throw, lancerà un'eccezione, che verrà affidata al metodo chiamante e salirà a cascata fino al metodo main(), se non viene intercettata prima.

Quando lanciamo un'eccezione con throw in un metodo e richiamiamo tale metodo in un'altra parte del codice, l'eccezione va catturata e gestita. Si può fare nei seguenti due modi:

1. Inserendo il blocco try – catch dove viene chiamato il metodo in cui è stata lanciata l'eccezione con throw:

```
public class Throw1 {  
  
    public static void main(String[] args) {  
        Throw1 t = new Throw1();  
        t.eseguiTesto(null);  
        System.out.println("sono l'istruzione dopo la chiamata al metodo eseguiTesto()  
nel main()");  
    }  
  
    public void stampaTesto(String testo) throws CorsoJavaException {  
        if(testo == null){  
            throw new CorsoJavaException();  
        }  
    }  
  
    public void eseguiTesto(String testo){  
        try {  
            stampaTesto(testo);  
        } catch (CorsoJavaException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

L'output di questo codice è il seguente:

```
eccezione.CoroJavaException: ATTENZIONE: nel progetto Corso_JAVA_EE si e' verificato  
un errore!!!  
    at eccezione.Throw1.stampaTesto(Throw1.java:13)  
    at eccezione.Throw1.eseguiTesto(Throw1.java:19)  
    at eccezione.Throw1.main(Throw1.java:7)  
sono l'istruzione dopo la chiamata al metodo eseguiTesto() nel main()
```

2. Utilizzando la keyword throws per posticipare la cattura e la gestione dell'eccezione alla prossima chiamata del metodo in cui si sarebbe dovuta trattare l'eccezione:

```
public class Throw2 {  
  
    public static void main(String[] args) {  
        Throw2 t = new Throw2();  
        try {  
            t.eseguiTesto(null);  
        } catch (CorsoJavaException e) {  
            e.printStackTrace();  
        }  
        System.out.println("sono l'istruzione dopo la chiamata al metodo eseguiTesto()  
nel main()");  
    }  
  
    public void stampaTesto(String testo) throws CorsoJavaException {  
        if(testo == null){  
            throw new CorsoJavaException();  
        }  
    }  
  
    public void eseguiTesto(String testo) throws CorsoJavaException {  
        stampaTesto(testo);  
    }  
}
```

L'output di questo codice è il seguente:

```
eccezione.CoroJavaException: ATTENZIONE: nel progetto Corso_JAVA_EE si e' verificato  
un errore!!!  
    at eccezione.Throw2.stampaTesto(Throw2.java:17)  
    at eccezione.Throw2.eseguiTesto(Throw2.java:22)  
    at eccezione.Throw2.main(Throw2.java:8)  
sono l'istruzione dopo la chiamata al metodo eseguiTesto() nel main()
```

Si deve porre attenzione, però, al fatto che se utilizziamo la keyword throws anche sul metodo main(), l'eccezione verrà mandata alla JVM, che di fatto non gestirà l'eccezione. Per cui in questa casistica il programma si arresta appena si verifica l'eccezione e, nel nostro esempio, la stringa "sono l'istruzione dopo la chiamata al metodo eseguiTesto() nel main()" non comparirà in console perché non verrà eseguita, in quanto il programma si è arrestato prima. Quindi, detto ciò, è consigliato non utilizzare la keyword throws sul metodo main(), altrimenti le eccezioni non verranno mai gestite.

ARRAY, LISTE E COLLECTION, CLASSI WRAPPER, AUTOBOXING E UNBOXING

LISTE

Una lista è semplicemente un elenco di oggetti. Alcuni esempi di liste sono:

- i programmi televisivi del giorno;
- le città di una nazione;
- le nazioni;
- la lista della spesa;
- le auto che attraversano il casello autostradale;
- gli studenti immatricolati;
-

Generalmente le liste, nell'ambito del software, vengono utilizzate quando vengono effettuate le ricerche all'interno del database.

Abbiamo due tipi di liste:

1. Liste statiche, che hanno una lunghezza predefinita e non cambia nel tempo.

0	1	2	3	4	5
---	---	---	---	---	---

2. Liste dinamiche, che hanno una lunghezza iniziale non definita e che può cambiare nel tempo.

0	1	2	3	4	5	...	n
---	---	---	---	---	---	-----	---

Per la gestione delle liste, JAVA mette a disposizione 2 tipi di strutture dati:

1. Array, per gestire le liste statiche.
2. Collection, per gestire le liste dinamiche.

MAPPE

Una mappa è una collezione di oggetti identificati da una chiave univoca.

Key	Value
K1	E1
K2	E2
K3	E3
K4	E4
K5	E5

Ogni elemento (**value**) viene salvato nella mappa e viene associato ad una chiave (**key**).

Per accedere ad un elemento presente nella mappa, posso utilizzare la chiave associata.

Per esempio, immaginiamo di avere una lista di utenti. Una chiave univoca per definire una mappa di utenti può essere il codice fiscale. Quindi, possiamo creare una mappa che ha come chiave il codice fiscale e come valore associato ad ogni codice fiscale ha un'istanza dell'oggetto utente, che rappresenta la persona.

Lo scopo principale delle mappe è quello di gestire in maniera più efficace l'accesso agli elementi. Infatti, attraverso la chiave è possibile cercare un elemento, piuttosto che iterare una lista.

Per gestire le mappe, JAVA mette a disposizione una serie di classi:

- HashMap;
- Hashtable;
- Properties;
- TreeMap.

Queste classi si trovano nel package `java.util`.

ARRAY

Un array è un contenitore di elementi, che possono essere sia dati primitivi che complessi. Le caratteristiche di un array sono:

- ha una lunghezza fissa;
- la lunghezza viene specificata in fase di inizializzazione;
- gli elementi presenti all'interno dell'array devono essere tutti dello stesso tipo;
- il primo elemento dell'array si trova alla posizione 0, mentre l'ultimo alla posizione n-1, dove n è la lunghezza dell'array.

Array di 10 elementi	0	1	2	3	4	5	6	7	8	9

La sintassi per la definizione di un array è la seguente:

TipoDiDato[] nomeVariabile;

Si specifica il tipo di dato seguito da apertura e chiusura delle parentesi quadre.

L'inizializzazione si effettua con la keyword new:

nomeVariabile = new TipoDiDato[n];

In fase di inizializzazione è necessario specificare la lunghezza n dell'array.

La lunghezza è un numero intero positivo.

Esempi

```
int[] arrayDiInteri = new int[5];      /* definizione ed inizializzazione di un array di interi */  
Prodotto[] arrayDiProdotti;           /* definizione di un array di oggetti di tipo Prodotto */  
arrayDiProdotti = new Prodotto[4];     /* inizializzazione dell'array di oggetti di tipo Prodotto */
```

Per conoscere la lunghezza di un array è possibile utilizzare l'attributo length.

```
Type[] nomeVariabile = new Type[n];  
nomeVariabile.length restituisce come valore "n".
```

Per accedere all'iesimo elemento si deve fare:

nomeVariabile[indice]

Per accedere a tutti gli elementi di un array (in lettura e scrittura), possiamo utilizzare un ciclo for che parte da 0 a n-1, dove n è la lunghezza dell'array. In questo modo, all'interno del ciclo for, la variabile i potrà essere utilizzata all'interno delle parentesi quadre per scrivere o leggere all'iesimo elemento dell'array.

Possiamo anche creare, inizializzare e riempire un array nella stessa istruzione:

```
Type[] nomeVariabile = new Type[]{  
    new Type(...),  
    new Type(...),  
    new Type(...)  
};
```

Esempio 3

```
public Prodotto[] generaArrayDiProdotto() {  
    Prodotto[] prodotti = new Prodotto[] {  
        new Prodotto("Prodotto 1", "Produttore Prodotto 1"),  
        new Prodotto("Prodotto 2", "Produttore Prodotto 2"),  
        new Prodotto("Prodotto 3", "Produttore Prodotto 3")  
    };  
    return prodotti;  
}
```

L'array è costituito da 3 elementi.

quindi **prodotti.length = 3**

Vediamo un esempio con gli array:

```
import programmazione_a_oggetti_esempio.gestionale.Smartphone;  
  
public class Array {  
  
    public static void main(String[] args) {  
        Array array = new Array();  
        array.creaArray();  
    }  
  
    public void creaArray(){  
  
        int[] test; /*dichiarazione dell'array test*/  
        test = new int[5]; /*inizializzazione dell'array test*/  
  
        int[] numeri = new int[10]; /*definizione e inizializzazione array di elementi  
        di tipo intero*/  
  
        /*Accesso in scrittura all'array di interi in ogni singola posizione*/  
        /*#####*/  
        numeri[0] = 10;  
        numeri[1] = 11;  
        numeri[2] = 12;  
        numeri[3] = 13;  
        numeri[4] = 14;  
        numeri[5] = 15;  
        numeri[6] = 16;  
        numeri[7] = 17;  
        numeri[8] = 18;  
        numeri[9] = 19;  
        /*#####*/
```

```

/*Accesso in scrittura all'array di interi mediante il for*/
//****************************************************************************
for(int i = 0; i < numeri.length; i++){
    numeri[i] = 10+i;
}
//****************************************************************************

Smartphone[] prodotti = new Smartphone[3];/*definizione e inizializzazione
array di oggetti di tipo Smartphone*/

/*Accesso in scrittura all'array di Smartphone in ogni singola posizione*/
//****************************************************************************
prodotti[0] = new Smartphone();
prodotti[0].setMarca("Apple");
prodotti[0].setNome("Iphone");
/*...*/

prodotti[1] = new Smartphone();
prodotti[1].setMarca("Samsung");
prodotti[1].setNome("Note");
/*...*/

prodotti[2] = new Smartphone();
prodotti[2].setMarca("Huawai");
prodotti[2].setNome("P8Lite");
/*...*/
//****************************************************************************

/*Accesso in lettura all'array di Smartphone mediante il for*/
//****************************************************************************
for (int i = 0; i < prodotti.length; i++){
    Smartphone sm = prodotti[i];

    System.out.println(sm.getMarca() + " " + sm.getNome());
}

//****************************************************************************

/*Accesso in lettura all'array di Smartphone mediante il foreach*/
//****************************************************************************
for (Smartphone sm : prodotti){
    System.out.println(sm.getMarca() + " " + sm.getNome());
}
//****************************************************************************

}
}

```

ARRAY MULTIDIMENSIONALI

È possibile in JAVA anche creare array di array, ovvero array multidimensionali. Gli array multidimensionali sono utilizzati per gestire matrici.

La sintassi per la definizione di un array multidimensionale è la seguente:

```
TipoDiDato[][]...[] nomeVariabile = new TipoDiDato[n][m]...[k];
```

```
Array bidimensionale: int[][] nomeVariabile = new int[n][m];
```

```
Array tridimensionale: int[][][] nomeVariabile = new int[n][m][k];
```

```
...
```

Le matrici sono strutture che hanno n righe ed m colonne, come una tabella.

Vediamo un esempio:

```
public class ArrayMultidimensionale {  
  
    public static void main(String[] args) {  
        ArrayMultidimensionale arrayMultidimensionale = new ArrayMultidimensionale();  
  
        arrayMultidimensionale.creaArrayMultidimensionale();  
    }  
  
    public void creaArrayMultidimensionale(){  
  
        int[][] matrice = new int[10][10]; /*dichiarazione matrice 10x10, ossia 10  
righe e 10 colonne*/  
  
        /*Accesso in scrittura all'array multidimensionale con il ciclo for*/  
        /*#####*/  
        for (int i = 0; i < matrice.length; i++){ /*iterazione sulle righe della  
matrice*/  
            for (int j = 0; j < matrice[i].length; j++){ /*iterazione sulle colonne  
della matrice*/  
                matrice[i][j] = i + j;  
            }  
        }  
        /*#####*/  
  
        /*Accesso in lettura all'array multidimensionale con il ciclo for*/  
        /*#####*/  
        for (int i = 0; i < matrice.length; i++){ /*iterazione sulle righe della  
matrice*/  
            for (int j = 0; j < matrice[i].length; j++){ /*iterazione sulle colonne  
della matrice*/  
                System.out.println(matrice[i][j]);  
            }  
        }  
        /*#####*/  
    }  
}
```

INTERFACCE COLLECTION E LIST

Abbiamo visto precedentemente che gli array si utilizzano per la gestione delle liste statiche.

Quando, invece, abbiamo a che fare con liste dinamiche, ovvero liste di cui non sappiamo inizialmente la dimensione, possiamo utilizzare degli strumenti fatti apposta per la gestione di questi tipi di liste. Tali strumenti vengono chiamati collection.

Le collection sono un set di interfacce e classi, messe a disposizione da JAVA, che consentono di gestire oggetti di lunghezza variabile.

Le classi e le interfacce collection si trovano nel package `java.util`.

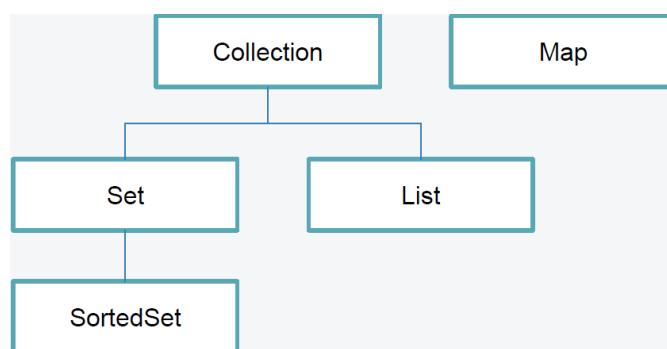
Tra i metodi per la manipolazione delle collection abbiamo:

- metodi per la ricerca di elementi all'interno di una collection;
- metodi per l'ordinamento di elementi all'interno di una collection.

Le interfacce principali sono:

- `Collection`;
- `Map`.

Le interfacce `Set` e `List` estendono `Collection`



INTERFACCIA COLLECTION

L'interfaccia `Collection` è l'interfaccia più generica. Da questa interfaccia derivano poi un set di classi, che implementano le proprietà e le caratteristiche di questa interfaccia e consentono di gestire appunto le liste.

L'interfaccia `Collection`:

- non definisce l'ordine in cui vengono memorizzati gli elementi;
- non definisce se ci possono essere elementi duplicati;
- non può contenere tipi primitivi, ma solamente oggetti. Per inserire tipi primitivi, è necessario effettuare il boxing.

INTERFACCIA LIST

L'interfaccia List estende l'interfaccia Collection e, rispetto a Collection, ha queste caratteristiche:

- gli oggetti vengono ordinati in base all'inserimento;
- può contenere duplicati;
- consente di aggiungere elementi specificando l'indice (ad esempio è possibile inserire un elemento nella posizione 5);
- consente di accedere agli elementi specificando l'indice.

CLASSI ARRAYLIST, HASHMAP E PROPERTIES

CLASSE ARRAYLIST

La classe ArrayList è l'implementazione dell'interfaccia List.

I costruttori della classe ArrayList sono 2:

- `ArrayList<E>():` serve per creare un'istanza della classe ArrayList vuota, ovvero senza specificare la capacità iniziale;
- `ArrayList<E>(int capacitaIniziale):` serve per creare un'istanza della classe ArrayList in cui viene specificata la capacità iniziale. In seguito, il valore della capacità iniziale può variare nel tempo a seconda della quantità di elementi che deve contenere.

Il simbolo `<E>` indica il tipo di elementi che sono presenti all'interno della lista.

Esempi

```
ArrayList<Prodotto> lista = new ArrayList<Prodotto>();
```

Definiamo una lista di oggetti di tipo Prodotto

```
ArrayList<String> lista = new ArrayList<String>();
```

Definiamo una lista di oggetti di tipo String

I metodi della classe ArrayList sono:

- `Object get(int index):` restituisce l'elemento che si trova alla posizione index specificata;
- `Object set(int index, Object obj):` sostituisce l'elemento presente nella posizione index con l'elemento obj che stiamo passando in ingresso;
- `boolean add(Object obj):` aggiunge un nuovo elemento obj subito dopo l'ultimo elemento;
- `void add(int index, Object obj):` inserisce l'elemento obj nella posizione index e sposta tutti gli altri elementi dalla posizione index in poi, scalandoli di una posizione;
- `int size():` restituisce il numero di elementi nella lista;
- `boolean isEmpty():` restituisce true se la lista è vuota, false se la lista contiene almeno un elemento;

- `Object remove(int index)`: cancella l'elemento che si trova nella posizione index;
- `int indexOf(Object elem)`: restituisce la prima posizione nella lista dell'elemento elem passato, oppure -1 se l'elemento non esiste;
- `String toString()`: restituisce una stringa contenente un tutti gli elementi presenti nella lista;
- `void clear()`: elimina tutti gli elementi della lista, svuotandola;
- `T[] toArray(T[] a)`: converte la lista in un array.

Vediamo un esempio:

```
import programmazione_a_oggetti_esempio.Smartphone;

import java.util.ArrayList;
import java.util.List;

public class ArrayListEsempio {

    public static void main(String[] args) {
        ArrayListEsempio al = new ArrayListEsempio();
        al.esempioArrayList();
    }

    public void esempioArrayList(){
        /*Primo modo per istanziare un oggetto di tipo ArrayList*/
        List<String> lista = new ArrayList<String>();

        lista.add("Paolo");
        lista.add("Mario");
        lista.add("Chiara");
        lista.add("Marta");

        for (String nome: lista){
            System.out.print(nome + " "); //Paolo Mario Chiara Marta
        }
        System.out.println();
        System.out.println("-----");

        lista.add(2, "Valerio");

        for (String nome: lista){
            System.out.print(nome + " "); //Paolo Mario Valerio Chiara Marta
        }
        System.out.println();
        System.out.println("-----");

        lista.remove(2);

        for (String nome: lista){
            System.out.print(nome + " "); //Paolo Mario Chiara Marta
        }
        System.out.println();
        System.out.println("-----");

        lista.set(2, "Claudia");
    }
}
```

```

for (String nome: lista){
    System.out.print(nome + " "); //Paolo Mario Claudia Marta
}
System.out.println();
System.out.println("-----");

System.out.println(lista.size()); //4

System.out.println("-----");

System.out.println(lista.isEmpty()); //false

lista.clear();

System.out.println("-----");

System.out.println(lista.size()); //0

System.out.println("-----");

System.out.println(lista.isEmpty()); //true

lista.add("Paolo");
lista.add("Mario");
lista.add("Chiara");
lista.add("Marta");

System.out.println("-----");

System.out.println(lista.indexOf("Mario")); //1

System.out.println("-----");

System.out.println(lista.toString()); // [Paolo, Mario, Chiara, Marta]

System.out.println("-----");

/*Conversione da ArrayList a Array*/
String[] listaArray = lista.toArray(new String[lista.size()]);

for (String nome : listaArray) {
    System.out.print(nome + " "); //Paolo Mario Chiara Marta
}

/*Secondo modo per istanziare un oggetto di tipo ArrayList*/
ArrayList<String> lista2 = new ArrayList<String>();

List<Smartphone> smartphone = new ArrayList<Smartphone>();

/* Siccome non abbiamo definito un costruttore per la classe Smartphone, */
/* che specifica in ingresso più parametri per caratterizzare l'oggetto, */
/* dobbiamo fare nella seguente maniera per aggiungere oggetti di tipo */
Smartphone nell lista dello stesso tipo: */
Smartphone sm = new Smartphone();
sm.setMarca("Apple");
sm.setModello("iPhone");

smartphone.add(sm);

```

```

        Smartphone sm2 = new Smartphone();
        sm2.setMarca("Samsung");
        sm2.setModello("Galaxy");

        smartphone.add(sm2);

    }

}

```

CLASSE HASHMAP

La classe HashMap serve per la gestione delle mappe. Tale classe è un'implementazione dell'interfaccia Map.

La classe HashMap ha due costruttori:

- `HashMap<K,V>():` crea un'istanza della classe HashMap vuota;
- `HashMap<K,V>(int initialCapacity):` crea un'istanza della classe HashMap in cui viene specificata la capacità iniziale, quindi il numero di elementi iniziali che verranno aggiunti all'interno della mappa.

Il simbolo `<K,V>` indica la coppia di elementi chiave-valore. K indica il tipo di dato che avranno le varie chiavi, mentre V indica il tipo di dato che avranno gli elementi che aggiungeremo all'interno della mappa.

Esempi

```
HashMap<String, Prodotto> lista = new HashMap<String, Prodotto>();
```

Definiamo una mappa che ha come chiave una variabile di tipo String e come valore un oggetto di tipo Prodotto

I metodi principali sono:

- `Object get(Object key)` restituisce l'elemento associato alla chiave passata in ingresso;
- `Object put(Object key, Object value)` aggiunge un elemento value alla lista e lo associa alla chiave key. Ovviamente se la mappa contiene già questa chiave, il vecchio elemento verrà sostituito da quello nuovo. Il metodo ritorna null se la chiave non è presente, quindi se si tratta di un nuovo inserimento di un oggetto, oppure ritorna il precedente oggetto associato a key se questa chiave era già presente all'interno della mappa;
- `Object remove(Object key)` rimuove l'elemento associato a key. Il metodo ritorna l'elemento che era associato a key prima della rimozione;
- `int size()` specifica il numero di elementi presenti nella mappa;
- `boolean containsKey(Object key)` restituisce true se la chiave è presente nella mappa, false altrimenti;
- `void clear()` elimina gli elementi dalla mappa;

- `Set<Object> keySet()` restituisce un oggetto di tipo `Set`, che contiene tutte le chiavi presenti nella mappa;
- `boolean isEmpty()` restituisce true se la mappa è vuota, false altrimenti.

Facciamo un esempio:

```
import programmazione_a_oggetti_esempio.Smartphone;

import java.util.HashMap;
import java.util.Map;

public class HashMapEsempio {

    public static void main(String[] args) {

        HashMapEsempio m = new HashMapEsempio();
        m.esempioHashMap();
    }

    public void esempioHashMap(){

        /*Primo modo per istanziare un oggetto di tipo HashMap*/
        Map<String, Smartphone> mappa = new HashMap<String, Smartphone>();

        Smartphone sm = new Smartphone();
        sm.setMarca("Apple");
        sm.setModello("iPhone");

        mappa.put(sm.getModello(), sm);

        Smartphone sm2 = new Smartphone();
        sm2.setMarca("Samsung");
        sm2.setModello("Galaxy");

        mappa.put(sm2.getModello(), sm2);

        Smartphone sm3 = mappa.get("iPhone");
        System.out.println(sm3.getMarca() + " " + sm3.getModello()); //Apple iPhone
        System.out.println("-----");

        System.out.println(mappa.size()); // 2
        System.out.println("-----");

        mappa.clear();

        System.out.println(mappa.size()); // 0
        System.out.println("-----");

        Smartphone sm4 = new Smartphone();
        sm4.setMarca("Apple");
        sm4.setModello("iPhone");

        mappa.put(sm.getModello(), sm3);

        Smartphone sm5 = new Smartphone();
        sm5.setMarca("Samsung");
        sm5.setModello("Galaxy");
    }
}
```

```

mappa.put(sm5.getModello(), sm5);

mappa.remove("Galaxy");

System.out.println(mappa.size()); // 1
System.out.println("-----");

System.out.println(mappa.keySet());// [iPhone]
System.out.println("-----");

System.out.println(mappa.containsKey("iPhone"));// true

/*Secondo modo per istanziare un oggetto di tipo HashMap*/
HashMap<String, Smartphone> mappa2 = new HashMap<String, Smartphone>();

}
}

```

CLASSE PROPERTIES

Questa classe è molto importante perché ci consente di gestire i file di configurazione dei nostri software.

Per esempio, quando creiamo un software che accede ad un database, generalmente le informazioni relative al database (indirizzo IP, porta, username, password, nome del database e così via) le dobbiamo salvare in file esterni al software. La classe Properties carica questi file e li converte in una mappa.

Questi file di configurazione hanno come estensione .properties.

Vediamo un esempio:

config.properties

```

db-name=corso
db-address=127.0.0.1
db-user=root
db-password=test

```

PropertiesEsempio.java

```

import java.io.*;
import java.util.Properties;

public class PropertiesEsempio {

    public static void main(String[] args) {

        PropertiesEsempio pe = new PropertiesEsempio();
        pe.getConfig();

    }

    public void getConfig(){
        Properties p = new Properties();

```

```
InputStream is = null;
try {
    is = new FileInputStream(new File("C:\\\\Users\\\\spanico\\\\IdeaProjects\\\\
Corso_JAVA_EE\\\\src\\\\array_liste_collection_classe_wrapper_autoboxing_unboxing\\\\
config.properties"));
    /*Metodo per caricare il file properties se esiste*/
    p.load(is);

    /*db-name=corso
    db-address=127.0.0.1
    db-user=root
    db-password=test*/
    System.out.println(p.getProperty("db-name")); // corso

} catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
}
```

INTERFACCIA ITERATOR

L'interfaccia Iterator è importante perché consente di iterare le Collection.

Tale interfaccia si trova nel package `java.util` e consente di generare classi per l'iterazione degli elementi di una Collection.

I metodi definiti dall'interfaccia Iterator sono 3:

- `hasNext()` ritorna true fino a quando ci sono elementi della lista da scorrere. Questo metodo tipicamente viene usato all'interno di un `while` come condizione;
 - `next()` ritorna l'elemento successivo;
 - `remove()` rimuove dalla lista l'ultimo elemento ritornato.

Il metodo remove deve essere usato solo durante un ciclo sugli elementi, altrimenti avremo l'eccezione ConcurrentModificationException quando accederemo al successivo elemento della lista.

Le interfacce Collection, List, Set e le relative classi che le implementano contengono il metodo `Iterator()` che ritorna un oggetto di tipo `Iterator`.

CICLARE LE COLLECTION

Vediamo un esempio pratico:

```
import programmazione_a_oggetti_esempio.Smartphone;

import java.util.*;

public class CiclareCollection {

    public static void main(String[] args) {
        CiclareCollection cc = new CiclareCollection();
        cc.esempioIterazione();
    }

    public void esempioIterazione(){

        /* Dichiarazione lista e popolamento */

        List<String> lista = new ArrayList<String>();

        lista.add("Paolo");
        lista.add("Mario");
        lista.add("Chiara");
        lista.add("Marta");

        /* Metodo 1 di iterazione per liste: for */

        System.out.print("Risultato iterazione con for su lista = ");
        for(int i = 0; i < lista.size(); i++){
            String nome = lista.get(i);

            System.out.print(nome + " ");
        }

        System.out.println();
        System.out.println("-----");

        /* Metodo 2 di iterazione per liste: foreach */

        System.out.print("Risultato iterazione con foreach su lista = ");

        for (String nome : lista) {
            System.out.print(nome + " ");
        }

        System.out.println();
        System.out.println("-----");

        /* Metodo 3 di iterazione per liste: interfaccia Iterator */

        System.out.print("Risultato iterazione con interfaccia Iterator su lista = ");

        Iterator<String> it = lista.iterator();

        while(it.hasNext()){
            String nome = it.next();

            System.out.print(nome + " ");
        }
    }
}
```

```

}

System.out.println();
System.out.println("-----");

/* Dichiarazione mappa e popolamento */

Map<String, Smartphone> mappa = new HashMap<String, Smartphone>();

Smartphone sm = new Smartphone();
sm.setMarca("Apple");
sm.setModello("iPhone");

mappa.put(sm.getModello(), sm);

Smartphone sm2 = new Smartphone();
sm2.setMarca("Samsung");
sm2.setModello("Galaxy");

mappa.put(sm2.getModello(), sm2);

Set<String> chiavi = mappa.keySet();

/* Metodo 1 di iterazione per le chiavi delle mappe: foreach */

System.out.print("Risultato iterazione con foreach sulle chiavi di una mappa = ");

for (String string : chiavi){
    System.out.print(string + " ");
}

System.out.println();
System.out.println("-----");

/* Metodo 2 di iterazione per le chiavi delle mappe: interfaccia Iterator */

System.out.print("Risultato iterazione con interfaccia Iterator sulle chiavi di una mappa = ");

Iterator<String> it2 = chiavi.iterator();

while (it2.hasNext()){
    String chiave = it2.next();

    System.out.print(chiave + " ");
}

System.out.println();
System.out.println("-----");

}

```

CLASSI WRAPPER

Le classi Wrapper sono delle particolari classi che trasformano un tipo di dato primitivo in un oggetto di tipo equivalente.

I tipi di dato primitivo in JAVA non sono oggetti. Nel tempo, però, si è avuta la necessità di definire delle classi che rappresentassero i tipi di dato primitivi e da qui abbiamo avuto la nascita delle classi Wrapper.

Esempio

```
int var1 = 5; /* questa è una variabile di tipo primitivo int */  
Integer var2 = new Integer(5); /* questa è un'istanza della classe Integer */
```

Per ogni tipo primitivo esiste una classe Wrapper.

Tipo primitivo	Classe Wrapper
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Vediamo un esempio:

```
public class ClasseWrapperEsempio {  
  
    public static void main(String[] args) {  
  
        /* È solo un dato primitivo, per cui JAVA non gli mette a disposizione alcun  
metodo */  
        int a = 10;  
  
        /* È un oggetto di tipo intero, per cui JAVA gli mette a disposizione dei  
metodi */  
        Integer b = new Integer(10);  
  
        Integer c = new Integer(a);  
  
    }  
}
```

BOXING, AUTOBOXING E UNBOXING

Il boxing è una caratteristica di JAVA che consente di “inserire” un tipo primitivo nella relativa classe Wrapper.

L'autoboxing è una caratteristica di JAVA che consente di convertire un tipo primitivo in una classe Wrapper.

L'unboxing è una caratteristica di JAVA che consente di convertire una classe Wrapper in un tipo primitivo.

Esempi di boxing

```
Integer var1 = new Integer (35);
Double var2 = new Double (15.6d);
Boolean var3 = Boolean.parseBoolean("false");

int varInt = 5;
Integer varInteger = new Integer(varInt);
```

Esempi di autoboxing

```
Integer var4 = 35;
Double var5 = 15.6;
Boolean var6 = false;
```

Esempi di unboxing

```
Integer var4 = 35;
Double var5 = 15.6;
Boolean var6 = false;

int var7 = var4;
double var8 = var5;
boolean var9 = var6;
```

Ovviamente si può intuire che questi 3 concetti sono strettamente legati alle classi Wrapper.

INTERFACCE

Un'interfaccia è un componente JAVA che consente di specificare azioni comuni a più tipi di oggetti, senza implementarne la logica.

Vediamo quali sono le caratteristiche dell'interfaccia:

- l'interfaccia contiene solo la definizione dei metodi e non l'implementazione;
- l'interfaccia non contiene costruttori;
- l'interfaccia può contenere delle variabili. Queste variabili possono essere solamente costanti, quindi devono contenere il modificatore final;
- l'interfaccia si definisce attraverso la parola riservata interface;
- una classe che implementa un'interfaccia ha l'obbligo di implementarne tutti i metodi;
- una classe può ereditare solo una classe;
- una classe può implementare una o più interfacce per gestire l'ereditarietà multipla;
- un'interfaccia può ereditare una o più interfacce.

```
public class ClasseC extends ClasseA, ClasseB {  
    ...  
}  
public class ClasseC implements InterfacciaA, InterfacciaB {  
    ...  
}
```

NON SI PUO' FARE

SI PUO' FARE

Generalmente un'interfaccia si utilizza per definire dei metodi comuni a più classi, che non hanno alcuna relazione gerarchica tra di loro.

Vediamo un esempio:

GeneraDati.java

```
public interface GeneraDati {  
    public String generaXML();  
}
```

Prodotto.java

```
public class Prodotto implements GeneraDati{  
  
    private long id;  
    private String nome;  
    private String codice;  
    private double prezzo;  
  
    public long getId() {  
        return id;  
    }
```

```

}

public void setId(long id) {
    this.id = id;
}

public String getNome() {
    return nome;
}

public void setNome(String nome) {
    this.nome = nome;
}

public String getCodice() {
    return codice;
}

public void setCodice(String codice) {
    this.codice = codice;
}

public double getPrezzo() {
    return prezzo;
}

public void setPrezzo(double prezzo) {
    this.prezzo = prezzo;
}

@Override
public String generaXML() {
    String xml =
        "<prodotto>" +
        "  <codice>" + getCodice() + "</codice>" +
        "  <nome>" + getNome() + "</nome>" +
        "  <prezzo>" + getPrezzo() + "</prezzo>" +
        "</prodotto>";

    return xml;
}
}

```

FormaGeometrica.java

```

public class FormaGeometrica implements GeneraDati{

    private double lato1;
    private double lato2;
    private double lato3;
    private double lato4;
    private double lato5;

    public double getLato1() {
        return lato1;
    }
}

```

```
public void setLato1(double lato1) {
    this.lato1 = lato1;
}

public double getLato2() {
    return lato2;
}

public void setLato2(double lato2) {
    this.lato2 = lato2;
}

public double getLato3() {
    return lato3;
}

public void setLato3(double lato3) {
    this.lato3 = lato3;
}

public double getLato4() {
    return lato4;
}

public void setLato4(double lato4) {
    this.lato4 = lato4;
}

public double getLato5() {
    return lato5;
}

public void setLato5(double lato5) {
    this.lato5 = lato5;
}

public double calcolaPerimetro(){
    return 0; /* calcolo perimetro */
}

public double calcolaArea(){
    return 0; /* calcolo area */
}

@Override
public String generaXML() {
    return null;
}
}
```

CLASSI ASTRATTE

Una classe astratta è un elemento JAVA che consente, analogamente all’interfaccia, di dichiarare caratteristiche comuni fra più classi.

A differenza dell’interfaccia, la classe astratta può contenere anche la definizione e l’implementazione dei metodi, così come può contenere attributi e costruttori. Abbiamo visto che, invece, l’interfaccia può contenere solo la definizione dei metodi e variabili costanti.

Quello che non possiamo fare con una classe astratta, rispetto a una classe normale, è instanziarla con la keyword new.

Una classe astratta si definisce attraverso le parole riservate abstract class.

Tutti i metodi astratti devono essere preceduti dalla parola riservata abstract e sono tutti quei metodi che non hanno l’implementazione.

```
package it.corsi.java;
public abstract class FormaGeometrica implements ConvertiDati {
    abstract double calcolaPerimetro();
    abstract double calcolaArea();
    abstract void disegnaForma();
    public String toString() {
        return FormaGeometrica.class.getSimpleName();
    }
}
```

Una classe astratta si utilizza principalmente per definire, appunto, delle entità astratte (o di alto livello), da cui è possibile definire altre classi che hanno una relazione gerarchica con essa.

Esempio:

- le classi Quadrato e Rettangolo rappresentano due figure geometriche;
- su ciascuna forma geometrica è possibile calcolare l’area ed il perimetro;
- la classe astratta FormaGeometrica definisce i metodi per calcolare area e perimetro, che poi verranno implementati nelle classi Quadrato e Rettangolo.

```
public abstract class FormaGeometrica implements ConvertiDati {
    abstract double calcolaPerimetro();
    abstract double calcolaArea();
    abstract void disegnaForma();
    ...
}
```

Vediamo un esempio:

Prodotto.java

```
public abstract class Prodotto {

    private long id;
    private String nome;
    private String descrizione;
    private double prezzo;

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getDescrizione() {
        return descrizione;
    }

    public void setDescrizione(String descrizione) {
        this.descrizione = descrizione;
    }

    public double getPrezzo() {
        return prezzo;
    }

    public void setPrezzo(double prezzo) {
        this.prezzo = prezzo;
    }

    public abstract double calcolaSpedizione();

    public abstract int calcolaVendite();
}
```

Televisore.java

```
public class Televisore extends Prodotto{

    @Override
    public double calcolaSpedizione() {
        return 0;
    }

    @Override
    public int calcolaVendite() {
        return 0;
    }
}
```

Lavatrice.java

```
public class Lavatrice extends Prodotto{

    @Override
    public double calcolaSpedizione() {
        return 0;
    }

    @Override
    public int calcolaVendite() {
        return 0;
    }
}
```

CLASSE INNER, LOCALE E ANONIMA

La regola generale in JAVA dice che per ogni classe deve esistere un corrispondente file.java. Inoltre il nome della classe deve essere uguale al nome del file.

Esempio:

se ho la classe Prodotto, il nome del file deve essere Prodotto.java.

Inoltre, per creare istanze di una classe JAVA, la sintassi da utilizzare è:

```
NomeClasse var = new NomeClasse(...);
```

Fanno eccezione a questa regola generale di JAVA 3 tipi di classe:

- classe inner: è una classe definita all'interno di un'altra classe, fuori dai metodi. La classe inner può essere utilizzata in lettura anche fuori dalla classe che la contiene, ma le sue istanze possono essere create solamente all'interno dei metodi della classe che la contiene;
- classe locale: è una classe definita all'interno di un metodo di un'altra classe;
- classe anonima: è un costrutto particolare che consente di creare ed istanziare una classe, senza dover creare un file.java. Vengono generalmente utilizzate nell'implementazione delle interfacce grafiche. Sono classi che vengono definite, inizializzate e istanziate nello stesso momento.

Esempio classe inner:

```
class Appartamento {  
    private String indirizzo;  
    private List<Stanza> stanze;  
  
    public void aggiungiStanza(String nome, String piano) {  
        ...  
        Stanza s = new Stanza();  
        s.setNomeStanza(nome);  
        s.setPiano(piano);  
        stanze.add(s);  
    }  
  
    class Stanza {  
        private String nomeStanza;  
        private String piano;  
        ...  
    }  
    ...  
}
```

Esempio classe inner:

```
public class Test {  
    Appartamento app = new Appartamento();  
  
    app.setIndirizzo("Via test");  
    app.aggiungiStanza("salotto", "terra");  
    app.aggiungiStanza("bagno", "primo");  
    app.aggiungiStanza("camera da letto matrimoniale", "primo");  
  
    for(Stanza stanza : app.getStanze()) {  
        System.out.println(stanza.getNomeStanza() + " - " + stanza.getPiano());  
    }  
}
```

Esempio classe local:

```
class Appartamento {  
    private String indirizzo;  
    private List<Stanza> stanze;  
    private double metriQuadrati;  
  
    public void aggiungiStanza(String nome, String piano, double lato1, double lato2) {  
        class CalcolaMetriQuadrati {  
            public double calcola(double lato1, double lato2) {  
                return lato1*lato2;  
            }  
        }  
        ...  
        Stanza s = new Stanza();  
        s.setNomeStanza(nome);  
        stanze.add(s);  
        ...  
        CalcolaMetriQuadrati mq = new CalcolaMetriQuadrati();  
        this.aggiungiMetriQuadrati(mq.calcola(lato1, lato2));  
    }  
    ...  
}
```

Vediamo un altro esempio sulle classi inner e local:

Appartamento.java

```
import java.util.ArrayList;
import java.util.List;

public class Appartamento {

    private int piano;

    private List<StanzaStandard> stanzeSt;
    private List<StanzaInner> stanzeIn;

    public void aggiungiStanza(double mq, String accessori){
        /*All'interno di Appartamento è possibile istanziare oggetti della classe
        StanzaInner*/
        getStanzeIn().add(new Appartamento.StanzaInner(16,"angolo cottura"));

        /*Classe local*/
        class CalcolaPerimetro {

            public double calcola(double lato1, double lato2){
                return lato1 + lato2;
            }
        }

        CalcolaPerimetro cp = new CalcolaPerimetro();
        cp.calcola(3, 6);
    }

    public int getPiano() {
        return piano;
    }

    public void setPiano(int piano) {
        this.piano = piano;
    }

    public List<StanzaStandard> getStanzeSt() {
        if(stanzeSt == null){
            stanzeSt = new ArrayList<StanzaStandard>();
        }

        return stanzeSt;
    }

    public void setStanzeSt(List<StanzaStandard> stanzeSt) {
        this.stanzeSt = stanzeSt;
    }

    public List<StanzaInner> getStanzeIn() {
        if(stanzeIn == null){
            stanzeIn = new ArrayList<StanzaInner>();
        }

        return stanzeIn;
    }
}
```

```

}

public void setStanzeIn(List<StanzaInner> stanzeIn) {
    this.stanzeIn = stanzeIn;
}

/*Classe Inner*/
class StanzaInner {

    private double mq;
    private String accessori;

    public StanzaInner(double mq, String accessori) {
        this.mq = mq;
        this.accessori = accessori;
    }

    public double getMq() {
        return mq;
    }

    public void setMq(double mq) {
        this.mq = mq;
    }

    public String getAccessori() {
        return accessori;
    }

    public void setAccessori(String accessori) {
        this.accessori = accessori;
    }
}
}

```

StanzaStandard.java

```

public class StanzaStandard {
    private double mq;
    private String accessori;

    public StanzaStandard(double mq, String accessori) {
        this.mq = mq;
        this.accessori = accessori;
    }

    public double getMq() {
        return mq;
    }

    public void setMq(double mq) {
        this.mq = mq;
    }

    public String getAccessori() {
        return accessori;
    }
}

```

```
        public void setAccessori(String accessori) {
            this.accessori = accessori;
        }
    }
```

MainInnerLocal.java

```
import java.util.List;

public class MainInnerLocal {

    public static void main(String[] args) {

        Appartamento a = new Appartamento();

        a.setPiano(0);

        a.getStanzeSt().add(new StanzaStandard(30,"angolo cottura"));

        /*L'istruzione di seguito è un errore perchè non possiamo istanziare oggetti
        della classe StanzaInner all'esterno della classe Appartamento*/
        /*a.getStanzeIn().add(new Appartamento.StanzaInner(16,"angolo cottura"));
        //java: an enclosing instance that contains
        classe_inner_local_anonima.Appartamento.StanzaInner is required */

        a.aggiungiStanza(16,"angolo cottura");

        List<StanzaStandard> st = a.getStanzeSt();

        for(StanzaStandard stanzaStandard : st){
            System.out.println(stanzaStandard.getMq()); // 30.0
        }

        List<Appartamento.StanzaInner> st2 = a.getStanzeIn();

        for (Appartamento.StanzaInner stanzaInner : st2) {
            System.out.println(stanzaInner.getMq()); // 16.0
        }
    }
}
```

Esempio classe anonima:

```
interface ActionListener {  
    void actionPerformed(ActionEvent e);  
}  
  
class A {  
    public ActionListener getButtonListener(final JButton b) {  
        return new ActionListener() {  
            public void actionPerformed(ActionEvent e) {  
                // codice eseguito quando l'utente preme il pulsante  
            }  
        };  
    }  
}
```

Vediamo un altro esempio sulla classe anonima:

Azione.java

```
public interface Azione {  
  
    public void eseguiAzione();  
}
```

AzioneImpl.java

```
public class AzioneImpl implements Azione{  
  
    @Override  
    public void eseguiAzione() {  
        System.out.println("Sono nella classe AzioneImpl");  
    }  
}
```

MainAnonima.java

```
public class MainAnonima {  
  
    public static void main(String[] args) {  
  
        MainAnonima m = new MainAnonima();  
  
        Azione a1 = m.eseguiAzione();  
        a1.eseguiAzione(); //Sono nella classe AzioneImpl  
        Azione a2 = m.eseguiAzione2();  
        a2.eseguiAzione(); //Sono nella classe anonima Azione  
    }  
  
    public Azione eseguiAzione() {  
        return new AzioneImpl();  
    }  
}
```

```

public Azione eseguiAzione2(){

    /*Classe anonima*/
    return new Azione() {
        @Override
        public void eseguiAzione() {
            System.out.println("Sono nella classe anonima Azione");
        }
    };
}

```

APPROFONDIMENTO CLASSI INNER

E' possibile istanziare classi inner fuori dalla classe che la definisce?

La risposta è SI.

Riprendiamo l'esempio della video lezione:

```

public class Appartamento {
    ...

    class StanzaInner {
        private double mq;
        private String accessori;
        ...
    }
}

```

Come abbiamo visto, la classe StanzaInner, così com'è definita non può essere istanziata fuori dalla classe Appartamento, per cui, in una classe esterna (che ad es. chiamiamo Main) non potrò fare:

```

public class Main {
    public static void main(String[] args) {
        Appartamento a = new Appartamento();

        StanzaInner si = new StanzaInner(16, "angolo cottura");
    }
}

```

Per poter istanziare la classe StanzaInner all'esterno della classe Appartamento la soluzione è rendere la classe static, quindi avremo:

```

public class Appartamento {

    ...
    static class StanzaInner {
        private double mq;
        private String accessori;

        public StanzaInner(double mq, String accessori) {
            super();
            this.mq = mq;
            this.accessori = accessori;
        }
    ...
}

```

A questo punto possiamo istanziare la classe StanzaInner, utilizzando la seguente sintassi:

```
ClasseInner nomeVariabile = new ClassePrincipale.ClasseInner();
```

Quindi, tornando al nostro esempio, avremo:

```
public class Main {  
    public static void main(String[] args) {  
        Appartamento a = new Appartamento();  
        StanzaInner si = new Appartamento.StanzaInner(16, "angolo cottura");  
    ...  
}
```

Ecco il codice completo delle classi Appartamento e Main:

```
import java.util.ArrayList;  
import java.util.List;  
  
public class Appartamento {  
    private int piano;  
  
    private List<StanzaStandard> stanzeSt;  
    private List<StanzaInner> stanzeIn;  
  
    public void aggiungiStanza(double mq, String accessori) {  
        class CalcolaPerimetro {  
            public double calcola(double lato1, double lato2) {  
                return lato1+lato2;  
            }  
        }  
        CalcolaPerimetro cp = new CalcolaPerimetro();  
        cp.calcola(3, 6);  
  
        getStanzeIn().add(new StanzaInner(mq, accessori));  
    }  
  
    static class StanzaInner {  
        private double mq;  
        private String accessori;  
  
        public StanzaInner(double mq, String accessori) {  
            super();  
            this.mq = mq;  
            this.accessori = accessori;  
        }  
        /**  
         * @return the mq  
         */  
        public double getMq() {  
            return mq;  
        }  
        /**  
         * @param mq the mq to set  
         */  
        public void setMq(double mq) {  
            this.mq = mq;  
        }  
        /**
```

```

        * @return the accessori
    */
    public String getAccessori() {
        return accessori;
    }
    /**
     * @param accessori the accessori to set
     */
    public void setAccessori(String accessori) {
        this.accessori = accessori;
    }

}

/**
 * @return the piano
 */
public int getPiano() {
    return piano;
}
/**
 * @param piano the piano to set
 */
public void setPiano(int piano) {
    this.piano = piano;
}
/**
 * @return the stanzeSt
 */
public List<StanzaStandard> getStanzeSt() {
    if(stanzeSt == null) {
        stanzeSt = new ArrayList<StanzaStandard>();
    }

    return stanzeSt;
}
/**
 * @param stanzeSt the stanzeSt to set
 */
public void setStanzeSt(List<StanzaStandard> stanzeSt) {
    this.stanzeSt = stanzeSt;
}
/**
 * @return the stanzeIn
 */
public List<StanzaInner> getStanzeIn() {
    if(stanzeIn == null) {
        stanzeIn = new ArrayList<StanzaInner>();
    }

    return stanzeIn;
}
/**
 * @param stanzeIn the stanzeIn to set
 */
public void setStanzeIn(List<StanzaInner> stanzeIn) {
    this.stanzeIn = stanzeIn;
}
}

```

```
package it.corso.java.classinnerlocali;

import java.util.List;
import it.corso.java.classinnerlocali.Appartamento.StanzaInner;

public class Main {
    public static void main(String[] args) {
        Appartamento a = new Appartamento();

        a.setPiano(0);

        a.getStanzeSt().add(new StanzaStandard(16, "angolo cottura"));

        StanzaInner si = new Appartamento.StanzaInner(16, "angolo cottura");
        a.getStanzeIn().add(si);

        a.aggiungiStanza(30, "angolo cottura");

        List<StanzaStandard> st = a.getStanzeSt();

        for (StanzaStandard stanzaStandard : st) {
            System.out.println(stanzaStandard.getMq());
        }

        List<StanzaInner> st2 = a.getStanzeIn();

        for (StanzaInner stanzaInner : st2) {
            System.out.println(stanzaInner.getMq());
        }
    }
}
```

JAVA GENERICS

I generics sono stati introdotti a partire dalla versione 5 di JAVA.

I generics sono degli strumenti, messi a disposizione dal linguaggio di programmazione, che ci consentono di definire degli elementi parametrizzati, attraverso la notazione <Tipo>.

Vediamo cosa vuol dire definire elementi parametrizzati:

Prima di JAVA 5, quando creavamo le liste, avevamo la seguente sintassi:

```
List nomeVariabile = new ArrayList();
```

Questo tipo di sintassi non specificava cosa doveva contenere la lista al suo interno, quindi per accedere ad ogni suo elemento, dovevo fare il casting. Per risolvere questo problema, attraverso i generics è stata implementata una nuova modalità di definizione degli elementi parametrizzati:

```
List<Tipo> nomeVariabile = new ArrayList<Tipo>();
```

Vediamo un esempio:

```
import java.util.List;
import java.util.ArrayList;

public class MainGenerics {

    public static void main(String[] args) {

        /* senza generics */
        List lista1 = new ArrayList();

        /* senza generics posso aggiungere elementi di tipo diverso,
         * ma ciò può creare problemi */
        lista1.add("es 1");
        lista1.add("es 2");
        lista1.add(new Appartamento());

        String test1 = (String) lista1.get(0);

        /* con generics */
        List<String> lista2 = new ArrayList<>();

        lista2.add("es 3");
        lista2.add("es 4");

        String test2 = lista2.get(0);
    }
}
```

L'utilizzo dei generics offre una serie di vantaggi, in particolare:

- abbiamo una migliore gestione del type checking, cioè sappiamo il tipo di elementi che contiene una lista già in fase di compilazione ;
- ci consente di evitare il casting, che è un'operazione abbastanza pesante per JAVA.

ANNOTATIONS

Le annotations sono degli appunti che vengono messi all'interno del codice sorgente per specificare qualcosa in merito ad un attributo, o un metodo, o una classe.

JAVA mette già a disposizione delle annotations, ma le possiamo anche creare per conto nostro. Vediamone alcune tra quelle messe a disposizione da JAVA:

- `@Deprecated`: è utilizzata per specificare che l'elemento (variabile, metodo o classe) è deprecato.
Deprecato vuol dire che l'elemento è ancora mantenuto all'interno del progetto per garantire la retrocompatibilità, ma è comunque un elemento che nel futuro può essere eliminato e ne è sconsigliato l'uso all'interno di nuovi progetti.
- `@Override`: annota al compilatore il fatto che quel metodo effettua l'override di un metodo della classe padre o di un'interfaccia;
- `@SuppressWarnings`: è utilizzata per dire al compilatore di non scrivere delle indicazioni di warning in fase di esecuzione.

Vediamo un esempio:

Prodotto.java

```
public class Prodotto {

    private long id;
    private String nome;
    private String descrizione;
    private double prezzo;

    @Deprecated
    public double calcolaIva(double ivaperc){
        return getPrezzo() * ivaperc / 100;
    }

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getDescrizione() {
        return descrizione;
    }

    public void setDescrizione(String descrizione) {
        this.descrizione = descrizione;
    }
}
```

```
}

public double getPrezzo() {
    return prezzo;
}

public void setPrezzo(double prezzo) {
    this.prezzo = prezzo;
}
}
```

MainAnnotation.java

```
public class MainAnnotation {

    public static void main(String[] args) {

        Prodotto p = new Prodotto();
        p.calcolaIva(22);
    }
}
```

GESTIONE DEI FILE

INTRODUZIONE ALLA GESTIONE DEI FILE

Per lavorare con i file, JAVA mette a disposizione diverse classi che consentono di:

- creare file e directory;
- cercare file e directory;
- rinominare file e directory;
- cancellare file e directory;
- scrivere file e directory;
- leggere il testo contenuto in un file.

Quindi, a seconda delle classi messe a disposizione da JAVA, possiamo effettuare determinate operazioni.

Abbiamo già visto, nel caso delle hashmap, la classe Properties che prende in ingresso una classe FileInputStream per la lettura di un file di testo, in particolare di un file di configurazione.

Le classi principali per le gestione dei file si trovano nel package java.io (io sta per InputOutput) e sono:

- per la gestione
 - java.io.File
- per la scrittura
 - java.io.FileWriter
 - java.io.BufferedWriter
 - java.io.PrintWriter
- per la lettura
 - java.io.FileReader
 - java.io.BufferedReader

La classe File è la rappresentazione di un file o di una directory. Questa classe permette di creare, eliminare, rinominare e cercare un file vuoto o una directory (è possibile anche sottodirectory).

La classe FileWriter consente di scrivere un carattere per volta in un file di testo.

La classe BufferedWriter consente di scrivere i caratteri nei file in blocchi, quindi utilizza un buffer temporaneo per memorizzare una serie di caratteri e scriverli direttamente poi nel file. In questa maniera abbiamo delle performance migliori rispetto la classe FileWriter.

La classe PrintWriter consente di scrivere nel file del testo formattato.

La classe FileReader consente di leggere i caratteri contenuti in un file di testo, leggendoli un carattere per volta.

La classe BufferedReader, analogamente per quanto accade alla BufferedWriter, utilizza un buffer temporaneo per la lettura di caratteri. In questo modo abbiamo delle performance migliori nella lettura dei file.

CLASSE FILE

Il costruttore principale della classe File riceve in ingresso una variabile di tipo String, che deve contenere il path del file o della directory. Ad esempio:

- “C:/corsi/java/lezioneX”
- “C:/corsi/java/lezioneX/file1.txt”

La creazione di un’istanza della classe File non determina in automatico la creazione fisica del file o della directory sul disco fisso. Tale istanza è semplicemente la rappresentazione di un file o di una directory.

Per creare una directory o un file fisicamente è necessario utilizzare dei metodi messi a disposizione dalla classe File, in particolare il metodo createNewFile() per creare file e il metodo mkdir() per creare directory.

I metodi principali della classe File sono:

- exists(): restituisce true se il file o la directory esiste, false altrimenti;
- createNewFile(): crea un file sul disco nel path specificato;
- delete(): elimina il file o la directory dal disco;
- isFile(): restituisce true se il path specificato è un file, false altrimenti;
- isDir(): restituisce true se il path specificato è una directory, false altrimenti;
- mkdir(): crea una directory sul disco nel path specificato.

Vediamo un esempio:

```
import java.io.File;
import java.io.IOException;

public class MainFile {

    public static void main(String[] args) {
        MainFile m = new MainFile();

        String dir = "C:\\\\Users\\\\spanico\\\\IdeaProjects\\\\Corso_JAVA_EE\\\\src\\\\file\\\\directory_crea_dal_codice";
        m.creaDirectory(dir);
        m.creaFile(dir + "\\\\file_creato_dal_codice.txt");

        File d = new File(dir);
        System.out.println(d.isDirectory()); //true
        System.out.println(d.isFile()); //false

        File f = new File(dir + "\\\\file_creato_dal_codice.txt");
        System.out.println(f.isDirectory());//false
        System.out.println(f.isFile()); //true
```

```
/*Il metodo listFiles() ritorna un array di file presenti nella directory specificata*/
File[] files = d.listFiles();

for(File file2 : files){
    System.out.println(file2.getName());// file_creato_dal_codice.txt

    file2.delete();
}
}

public void creaFile(String path){
    File f = new File(path);

    if (!f.exists()){
        try {
            f.createNewFile();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

public void creaDirectory(String path){
    File d = new File(path);

    if (!d.exists()){
        d.mkdir();
    }
}
```

CLASSI PER SCRIVERE SU FILE

CLASSE FILEWRITER

La classe FileWriter mette a disposizione i seguenti metodi:

- write(String str), che prende in ingresso la stringa da scrivere e la inserisce nel file;
- close(), che chiude lo stream sul file e, prima di effettuare lo stream, invoca il metodo flush;
- flush(), che garantisce che tutto il testo sia stato scritto nel file.

Il costruttore della classe FileWriter prende in ingresso un oggetto di tipo File.

CLASSE BUFFEREDWRITER

Il funzionamento della classe BufferedWriter è analogo a quello della classe FileWriter, solo che la classe FileWriter abbiamo detto che scrive un singolo carattere per volta, mentre la classe BufferedWriter scrive in maniera più efficiente perché ha un buffer interno in cui accumula una serie di caratteri e poi fa lo stream all'interno del file.

Quando creiamo un'istanza della classe BufferedWriter, dobbiamo passare in ingresso al costruttore un'istanza della classe FileWriter.

I metodi principali della classe BufferedWriter sono:

- write(String str), che scrive la stringa str nel file;
- newLine(), che crea una nuova linea all'interno del file, quindi il testo che segue questa istruzione viene scritto su una nuova linea;
- close() e flush() li abbiamo già visti nella classe FileWriter.

ESEMPIO CON LE CLASSI PER LA SCRITTURA DI UN FILE

```
import java.io.*;

public class ScritturaFile {

    public static void main(String[] args) {
        String dir = "C:\\\\Users\\\\spanico\\\\IdeaProjects\\\\Corso_JAVA_EE\\\\src\\\\file\\\\
directory_creato_dal_codice\\\\";

        ScritturaFile sf = new ScritturaFile();
        try {
            sf.esempioFileWriter(dir + "file_creato_e_scritto_dal_codice_FileWriter",
"Lorem ipsum ...");

            sf.esempioBufferedWriter(dir +
"file_creato_e_scritto_dal_codice_BufferedWriter", "LALALALA");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```

}

public void esempioFileWriter(String filePath, String testo) throws IOException {

    File file = new File(filePath);

    if(!file.exists()) {
        file.createNewFile();
    }

    FileWriter fw = new FileWriter(file);

    fw.write(testo);
    fw.close();

    /* Nel caso volessimo gestire noi le eccezioni, invece di demandarle al metodo
     * con throws, possiamo fare anche in quest altro modo:

    if(!file.exists()) {
        try {
            file.createNewFile();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    FileWriter fw = null;

    try {
        fw = new FileWriter(file);
        fw.write(testo);
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        try {
            fw.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }/*
}
}

public void esempioBufferedWriter(String filePath, String testo) throws IOException
{
    File file = new File(filePath);

    if(!file.exists()) {
        try {
            file.createNewFile();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    BufferedWriter bw = null;

    try {

```

```

        bw = new BufferedWriter(new FileWriter(file));
        bw.write(testo);
        bw.newLine();
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        try {
            bw.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

CLASSI PER LEGGERE FILE

CLASSE FILEREADER

Il costruttore della classe FileReader prende in ingresso un'istanza della classe File, analogamente a quanto avviene per la classe FileWriter.

I metodi principali della classe FileReader sono:

- `read(char[])`, che legge un numero finito di caratteri, uno alla volta, e li memorizza nell'array di char passato in ingresso;
- `close()`, che chiude lo stream dal file.

CLASSE BUFFEREDREADER

La classe BufferedReader consente di leggere da un file in maniera più efficiente rispetto alla classe FileReader, avendo un buffer interno.

Quando creiamo un'istanza della classe BufferedReader, al costruttore dobbiamo passare un'istanza della classe FileReader.

I metodi principali della classe BufferedReader sono gli stessi della classe FileReader.

ESEMPIO CON LE CLASSI PER LA LETTURA DI UN FILE

```

import java.io.*;

public class LetturaFile {

    public static void main(String[] args) {
        String dir = "C:\\\\Users\\\\spanico\\\\IdeaProjects\\\\Corso_JAVA_EE\\\\src\\\\file\\\\
directory_creato_dal_codice\\\\file_creato_e_scritto_dal_codice_FileWriter";

        LetturaFile lw = new LetturaFile();
        lw.esempioFileReader(dir);
        lw.esempioBufferedReader(dir);
    }
}

```

```

}

public void esempioFileReader(String filePath) {
    File f = new File(filePath);
    if(f.exists()){
        FileReader fr = null;
        try {
            fr = new FileReader(f);

            /*numero massimo di caratteri*/
            char[] testo = new char[1024];

            try {
                /*fr.read(testo);*/

                /*Faccio in modo che ciò che viene Letto
                 * nel file venga scritto anche sulla console*/
                /*#####
                int size = fr.read(testo);
                for (int i = 0; i < size; i++){
                    System.out.print(testo[i]);
                }
                #####
            } catch (IOException e) {
                e.printStackTrace();
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } finally {
            try {
                fr.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}

public void esempioBufferedReader(String filePath) {
    File f = new File(filePath);
    if(f.exists()){
        BufferedReader br = null;
        try {
            br = new BufferedReader(new FileReader(f));

            /*numero massimo di caratteri*/
            char[] testo = new char[1024];

            try {
                /*fr.read(testo);*/

                /*Faccio in modo che ciò che viene Letto
                 * nel file venga scritto anche sulla console*/
                /*#####
                int size = br.read(testo);
                for (int i = 0; i < size; i++){
                    System.out.print(testo[i]);
                }
                #####
            } catch (IOException e) {
                e.printStackTrace();
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}

```

```
        } catch (IOException e) {
            e.printStackTrace();
        }
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } finally {
        try {
            br.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

CLASSE INPUTSTREAMREADER PER LEGGERE GLI INPUT DA TASTIERA

Quando premiamo un tasto della tastiera, è possibile recuperarne il valore corrispondente attraverso la variabile `in` della classe `System`.

La variabile `in` è di tipo `InputStream` ed è statica per cui è possibile invocarla direttamente tramite la classe `System` (cioè scrivendo `System.in`), analogamente a quanto facciamo per la variabile `out` (`System.out.println...`)

Per leggere lo stream presente nella variabile possiamo utilizzare due classi:

- java.io.InputStreamReader
 - java.util.Scanner

Vediamo un esempio:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Scanner;

public class ReadInputTastiera {
    public static void main(String[] args) {
        leggi1();
    }

    public static void leggi1() {
        System.out.println("Benvenuto nel programma...");
        String saluto = "Ciao ";

        try {
            System.out.println("Inserisci il tuo nome:");

            BufferedReader bufferRead = new BufferedReader(new
InputStreamReader(System.in));

            saluto += bufferRead.readLine();
        }
    }
}
```

```

        System.out.println("Inserisci il tuo cognome:");
        bufferRead = new BufferedReader(new InputStreamReader(System.in));
        saluto += " " + bufferRead.readLine();
        System.out.println(saluto);
    } catch(IOException e) {
        e.printStackTrace();
    }
}
}

```

Per leggere lo stream, la prima cosa da fare è creare un'istanza della classe InputStreamReader, passando in ingresso System.in.

`new InputStreamReader(System.in)`

A questo punto possiamo creare un'istanza della classe BufferedReader, passando in ingresso il nostro InputStreamReader.

`BufferedReader bufferRead = new BufferedReader(new InputStreamReader(System.in));`

Ogni volta che invochiamo il metodo `readline()` della classe BufferedReader, il software si interromperà in attesa dell'input da tastiera.

`saluto += bufferRead.readLine();`

CLASSE SCANNER PER LEGGERE GLI INPUT DA TASTIERA

Vediamo un esempio:

```

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Scanner;

public class ReadInputTastiera {
    public static void main(String[] args) {
        leggi2();
    }

    public static void leggi2() {
        System.out.println("Benvenuto nel programma...");
        String saluto = "Ciao ";

        System.out.println("Inserisci il tuo nome:");
        Scanner scanIn = new Scanner(System.in);
        saluto += scanIn.nextLine();

        System.out.println("Inserisci il tuo cognome:");
        saluto += " " + scanIn.nextLine();

        scanIn.close();
    }
}

```

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

Per leggere lo stream, la prima cosa da fare è creare un'istanza della classe Scanner, passando in ingresso System.in.

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

Ogni volta che invochiamo il metodo nextLine() della classe Scanner, il software si interromperà in attesa dell'input da tastiera.

```
saluto += scanIn.nextLine();
```

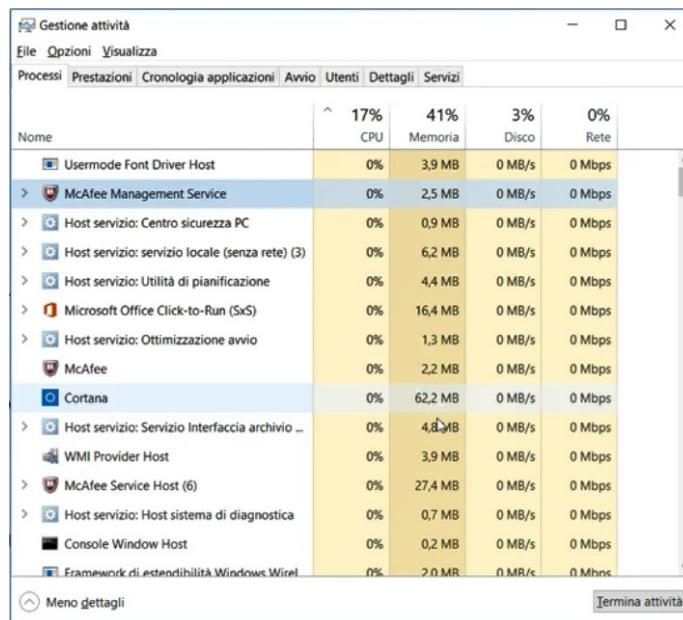
La lettura degli input da tastiera con la classe Scanner è ancora più semplice rispetto all'utilizzo della classe InputStreamReader.

THREAD E CONCORRENZA

PROCESSO

Prima di capire cosa è un thread, dobbiamo definire cosa è un processo.

Quando installiamo su un dispositivo un software, copiamo i file compilati del software stesso all'interno del dispositivo, indipendentemente dal tipo di sistema operativo che vi è installato sopra. Il software in stato di esecuzione genera un processo, che avrà un ID univoco e occuperà una porzione di RAM dove salverà le sue informazioni. Quindi, un processo può essere definito come un'istanza del software che contiene le istruzioni e i dati che vengono elaborati durante l'esecuzione del software. Un esempio lo possiamo vedere sulla gestione attività del PC, dove sono elencati tutti i processi attivi in quel dato momento:



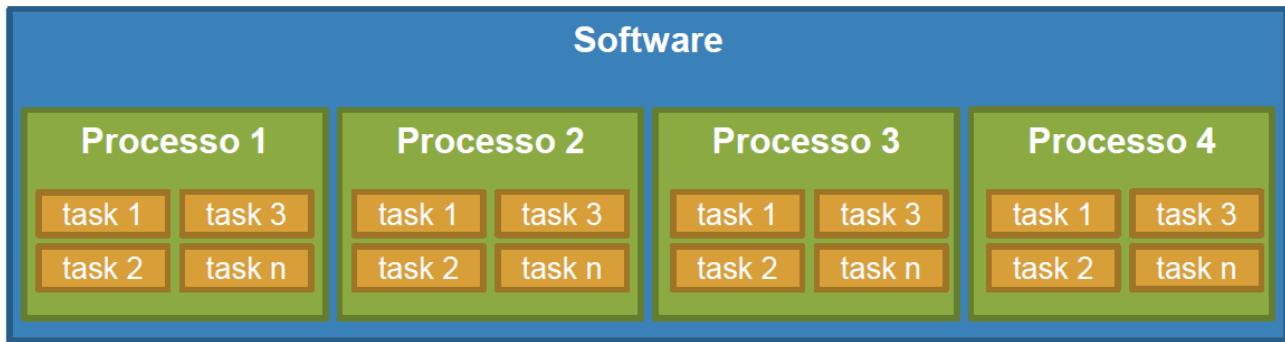
Siccome i sistemi operativi sono multitasking, possiamo eseguire più processi contemporaneamente e il sistema operativo assegnerà la CPU all'i-esimo processo, a seconda delle necessità.

THREAD

Quando abbiamo la necessità di suddividere un processo in più sottoprocessi (o task), i quali devono essere indipendenti tra loro, a quel punto parliamo di thread. Un thread è un sottoprocesso che ha vita propria e svolge un determinato compito. Più thread possono essere eseguiti in maniera concorrente, a seconda delle necessità.

Quindi, in linea generale lo scopo dei thread è quello di dividere un processo in tante piccole parti, dove ogni parte lavora in maniera autonoma e deve gestire un evento o una risorsa.

Vediamo un esempio:



In questo esempio abbiamo il nostro software e 4 processi, che sono intesi come 4 istanze dello stesso software. Il numero di processi equivale al numero di volte che l'utente avvia lo stesso software. Ogni processo avrà i suoi task per la gestione delle singole attività e dei singoli eventi.

Esempi di utilizzo dei thread sono:

Nel caso del browser web:

- un thread che si occupa di scrivere il testo e visualizzarlo a video;
- un thread che si occupa di effettuare la ricerca.

Nel caso del server web:

- un thread che si occupa di accettare le richieste e creare altri thread che le gestiscono;
- un thread che si occupa di gestire una richiesta.

Nel caso di word:

- un thread che si occupa di scrivere i comandi digitati dall'utente;
- un thread che si occupa di cercare i sinonimi e gli errori di scrittura;
- un thread che si occupa di visualizzare gli errori di ortografia.

CONCORRENZA

Ovviamente anche il sistema operativo stesso è composto da diversi software. Se il dispositivo non fosse in grado di gestire l'accesso concorrente a questi programmi, potremmo eseguire un solo programma alla volta. Ad esempio, non potremmo navigare su internet e al tempo stesso guardare un video, scrivere un documento, ascoltare musica e così via.

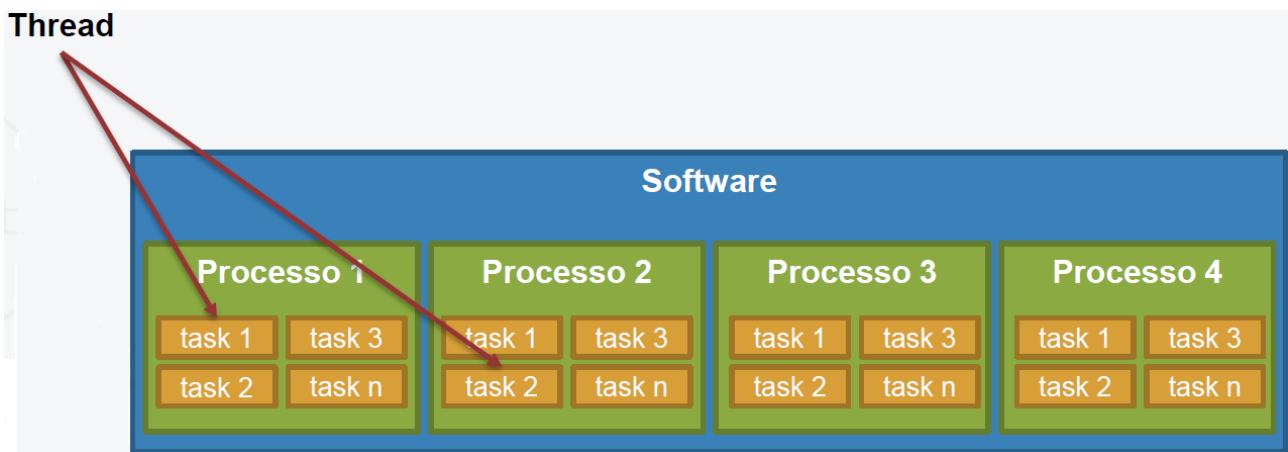
Si capisce a questo punto che la gestione della concorrenza diventa fondamentale, perché consente di sfruttare tutte le risorse hardware a disposizione per l'esecuzione di più software contemporaneamente.

Quindi, la concorrenza è un insieme di concetti che consentono di rappresentare e descrivere l'esecuzione di due o più processi in maniera simultanea o non simultanea, a seconda della tipologia di architettura su cui stiamo eseguendo un software. Due o più processi, quindi due o più istanze di diversi software o dello stesso software, sono in esecuzione concorrente se vengono eseguiti in parallelo. A seconda del tipo di architettura, abbiamo 2 tipi di parallelismo:

- il parallelismo reale avviene quando più processi sono attivi su un dispositivo dotato di più processori, dove ogni processore prende in carico un processo, oppure quando più processi sono attivi su più dispositivi indipendenti tra loro e distribuiti.
- Il parallelismo apparente avviene quando più processi sono attivi su un dispositivo dotato di un processore.

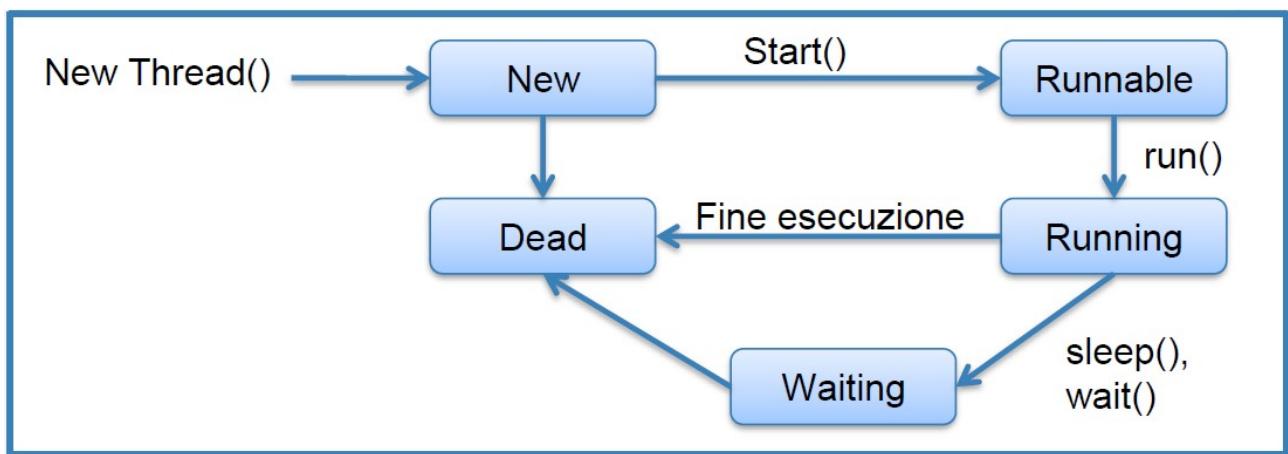
CICLO DI VITA DI UN THREAD

Un thread, dal punto di vista del processo, abbiamo già visto essere un sottoprocesso (o task), quindi un componente di un processo più ampio e complesso che si occupa di gestire una risorsa o un evento.



Un thread ha un suo ciclo di vita, che comprende i seguenti stati:

- New
- Runnable
- Running
- Waiting
- Dead



Il primo stato in cui si trova un thread è New quando viene istanziato, poi abbiamo Runnable quando viene avviato, Running quando viene eseguito. Arrivato allo stato di Running, un thread può arrivare a Waiting, oppure a Dead, a seconda del caso se vogliamo sospendere o terminare il suo ciclo di vita.

THREAD PRIORITY

La priority (o priorità) è l'informazione che indica allo scheduler il livello di importanza di un thread rispetto agli altri. Lo scheduler è un software che si occupa di gestire l'esecuzione dei thread durante l'esecuzione di un processo.

Facendo attenzione, però, su un concetto, non tutti i sistemi operativi garantiscono che l'ordine di esecuzione dei thread è determinato sulla base della thread priority che abbiamo stabilito. Tuttavia, se un certo numero di thread sono bloccati e in attesa di essere eseguiti, il primo che verrà sbloccato dallo scheduler sarà quello che ha priorità maggiore. In questo modo evitiamo la situazione che si chiama starvation, ovvero l'impossibilità di ottenere risorse da parte di un processo, perché tutti i task sono bloccati.

La priority di un thread in JAVA va da 1 a 10, dove la priorità minore è definita MIN_PRIORITY(1), la priorità maggiore è MAX_PRIORITY(10) e la priorità normale, che è anche quella di default, è NORM_PRIORITY(5).

CREARE UN THREAD IN JAVA

Per creare un thread in JAVA abbiamo due possibilità:

- creare una classe che estende la classe Thread
- creare una classe che implementa l'interfaccia Runnable

Ovviamente, la classe Thread implementa l'interfaccia Runnable.



Se creiamo un thread attraverso la classe Thread, la nostra classe deve ereditare attraverso la keyword `extends` la classe Thread. Questo permetterà alla nostra classe di ereditare i metodi della classe Thread, in particolare quello più importante di tutti che è il metodo `run()`. Il metodo `run()` deve essere riscritto nella nostra classe, cioè ci verrà richiesto di fare l'override di tale metodo. Il metodo `run()` contiene le istruzioni che devono essere eseguite dal thread. Il codice presente nel metodo `run()` viene eseguito in maniera concorrente ad altri thread presenti in un programma.

Vediamo un primo esempio di thread in JAVA:

```
public class EsempioThread extends Thread {  
    /*Tutto quello che è il blocco di codice che dobbiamo eseguire,  
     deve essere messo all'interno del metodo run()  
     */  
    @Override  
    public void run() {  
        System.out.println("sono un thread");  
    }  
  
    class MainThread{  
        public static void main(String[] args) {  
            EsempioThread et = new EsempioThread();  
  
            //Per eseguire il thread, dobbiamo invocare il metodo start()  
            et.start();//sono un thread  
        }  
    }  
}
```

Nell'esempio, l'esecuzione del metodo `start()` nel `main()` ha causato l'invocazione del metodo `run()`.

Quindi, per creare un thread dobbiamo:

- creare un'istanza della nostra classe che estende Thread;
- invocare il metodo `start()` che si occuperà, dopo la configurazione del thread, di invocare il metodo `run()`.

Altri metodi utili che si ereditano dalla classe Thread sono:

- `yield()` che suggerisce allo scheduler di liberare la CPU e renderla disponibile, ovviamente, ad altri thread;

- sleep() che mette in standby il thread per un determinato numero di millisecondi.

Possiamo eventualmente creare un thread anche utilizzando direttamente l'interfaccia Runnable. In questo caso la nostra classe deve implementare l'interfaccia Runnable. Questo secondo metodo viene utilizzato, piuttosto che utilizzare la classe Thread, quando la nostra classe estende già un'altra classe e, siccome non possiamo usare l'ereditarietà multipla con l'attributo extends, possiamo utilizzare l'interfaccia Runnable che ci consentirà comunque di implementare un thread. Ovviamente, implementando l'interfaccia, dobbiamo implementare il metodo run(), che è definita all'interno dell'interfaccia Runnable. Per creare un thread in questo caso dobbiamo utilizzare il costruttore della classe Thread, che prende in ingresso come parametro l'istanza della nostra classe che implementa Runnable. Vediamo un esempio:

```
public class EsempioRunnable implements Runnable{
    @Override
    public void run() {
        System.out.println("sono un thread runnable");
    }
}

class MainRunnable {
    public static void main(String[] args) {
        /*Per eseguire questo thread, dobbiamo creare un'istanza della
        classe Thread, passando in ingresso un'istanza della nostra
        classe che implementa l'interfaccia Runnable*/
        Thread t = new Thread(new EsempioRunnable());

        t.start(); //sono un thread runnable
    }
}
```

MULTITHREADING

Multithreading vuol dire eseguire contemporaneamente più thread appartenenti allo stesso processo. Il multithreading può essere:

- collaborativo, ovvero i thread rimangono attivi fino a quando non terminano il task, oppure fino a quando non cedono le risorse occupate ad altri thread;
- preventivo, ovvero la macchina virtuale accede ad un thread attivo e lo controlla attraverso un altro thread.

Le specifiche JAVA stabiliscono che la JVM debba gestire i thread, utilizzando lo scheduling preemptive (o fixed-priority scheduling). Questo vuol dire che lo scheduler ha il compito di interrompere o ripristinare i thread a seconda del loro stato. Quindi, in base in cui si trovano i thread, lo scheduler può attivarli o disattivarli.

Come abbiamo già detto, ogni esecuzione della JVM corrisponde ad un processo, mentre tutto quello che viene eseguito dalla JVM corrisponde ad un thread.

Chiaramente il multithreading ha ragione di esistere perché, se eseguiamo le operazioni in parallelo, ovviamente riusciamo a raggiungere un risultato in maniera più rapida. Oltre a questo, un'altra caratteristica dei software multithreading è che i thread possono scambiarsi informazioni tra loro ed accedere a risorse condivise, ad esempio un database condiviso tra thread, oppure una risorsa hardware.

Vediamo un esempio in codice di multithreading:

```
public class EsempioMultithreading extends Thread{
    @Override
    public void run(){

        System.out.println("Sono il thread " + getName());

        for (int i = 0; i < 10; i++){
            System.out.println(i);
        }
        /*Utilizzo il metodo sleep per dare un tempo di pausa
         tra una stampa di i e un'altra*/
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

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

        EsempioMultithreading em1 = new EsempioMultithreading();
        //setName() serve per poter assegnare un nome al thread
        em1.setName("Thread1");

        EsempioMultithreading em2 = new EsempioMultithreading();
        em2.setName("Thread2");

        EsempioMultithreading em3 = new EsempioMultithreading();
        em3.setName("Thread3");

        EsempioMultithreading em4 = new EsempioMultithreading();
        em4.setName("Thread4");

        EsempioMultithreading em5 = new EsempioMultithreading();
        em5.setName("Thread5");

        em1.start();
        em2.start();
        em3.start();
        em4.start();
        em5.start();
    }
}
```

In questo codice abbiamo 5 thread che ciclano la variabile i per 10 volte. Tutti e 5 i thread si eseguono in maniera concorrente tra di loro, richiedendo l'accesso alle risorse e, il più delle volte, interrompendosi tra di loro. Ragion per cui, eseguendo più volte questo codice, otterremo sempre un output diverso, perché cambierà sempre l'ordine di esecuzione dei thread e i momenti in cui uno sospenderà l'esecuzione dell'altro.

Vediamo un esempio dei tanti output che possiamo ottenere dal codice precedente:

Sono il thread Thread3

0

1

Sono il thread Thread1

0

1

2

3

4

5

6

7

8

9

Sono il thread Thread2

0

1

2

3

4

5

6

7

8

9

Sono il thread Thread4

0

1

2

3

4

5

6

7

8

9

Sono il thread Thread5

0

1

2

3

4

5

6

7

8

9

2

3

4

5

6

7

8

9

1

0

9

8

7

6

5

4

3

2

1

0

9

8

7

6

5

4

3

2

1

0

9

8

7

6

5

4

3

2

1

0

9

8

7

6

5

4

3

2

1

0

9

8

7

6

5

4

3

2

1

0

9

8

7

6

5

4

3

2

1

0

9

8

7

6

5

4

3

2

1

0

9

8

7

6

5

4

3

2

1

0

9

8

7

6

5

4

3

2

1

0

9

8

7

6

5

4

3

2

1

0

9

8

7

6

5

4

3

2

1

0

9

8

7

6

5

4

3

2

1

0

9

8

7

6

5

4

3

2

1

0

9

8

7

6

5

4

3

2

1

0

9

8

7

6

5

4

3

2

1

0

9

8

7

6

5

4

3

2

1

0

9

8

7

6

5

4

3

2

1

0

9

8

7

6

5

4

3

2

1

0

9

8

7

6

5

4

3

2

1

0

9

8

7

6

5

4

3

2

1

0

9

8

7

6

5

4

3

2

1

0

9

8

7

6

5

4

3

2

1

0

9

8

7

6

5

4

3

2

1

0

9

8

7

6

5

4

3

2

1

0

9

8

7

6

5

4

3

2

1

0

9

8

7

6

5

4

3

2

1

0

9

8

7

6

5

CONCORRENZA IN JAVA

La concorrenza è la possibilità di eseguire più task in parallelo. L'utilizzo della concorrenza è molto utile quando i processi possono essere parallelizzati, senza necessità di serializzarli. Ad esempio, immaginiamo che in un browser possiamo caricare contemporaneamente più pagine. Se il browser non fosse stato un software multithreading, non avremmo potuto effettuare questa operazione, ma avremmo dovuto aspettare il caricamento di una pagina per poterne richiedere un'altra.

In JAVA la concorrenza si implementa tramite tre strumenti:

- I Thread, disponibili dalla versione 1.0 di JAVA.
- Il framework Executor, disponibile dalla versione 1.5 di JAVA.
- Il framework Fork/Join, disponibile dalla versione 1.7 di JAVA.

CONCORRENZA CON L'UTILIZZO DEI THREAD

Per creare un thread è necessario, abbiamo già detto, creare la classe che estenda Thread, oppure che implementi Runnable, e che implementi in entrambi i casi il metodo run(). All'interno del metodo run() dobbiamo inserire la logica relativa al task da eseguire.

L'avvio del thread si ha con il metodo start(), mentre l'attesa del completamento del task (l'attesa dell'esecuzione completa del thread) si ha con il metodo join().

Se vogliamo passare dei parametri d'ingresso, siccome il metodo run() non accetta parametri d'ingresso ed è un metodo void (cioè non restituisce parametri), dobbiamo definire un costruttore personalizzato che riceva in ingresso i parametri necessari. A quel punto, possiamo lavorare con i parametri ricevuti in ingresso dal costruttore.

Facciamo un esempio in cui supponiamo di creare una classe che consenta di recuperare l'output (l'html) di una pagina web:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.MalformedURLException;
import java.net.URL;
import java.netURLConnection;

public class GetPaginaSito extends Thread {

    private String url;
    private String content;

    /*Creo il costruttore per poter passare un parametro in ingresso
     * alla logica all'interno di new()*/
    public GetPaginaSito(String url){
        super();
        this.url = url;
    }

    @Override
```

```

public void run() {
    /*Dentro il metodo run() implemento la logica che permette
    di invocare una URL e di recuperare le informazioni*/
    try {
        /*Sto passando la url in ingresso dal costruttore*/
        URL u = new URL(url);

        /*Creo la connessione*/
        URLConnection con = u.openConnection();

        /*Attraverso l'invocazione del metodo getInputStream(), recuperiamo
        l'output del nostro sito, quindi la nostra pagina html*/
        InputStream is = con.getInputStream();

        /*Scrivo il contenuto della pagina html in un file, mediante il
        metodo getString() della classe Utils, che prende in ingresso
        un InputStream e scrive in uno StringBuilder tutto l'output
        della nostra pagina web*/
        setContent(Utils.getString(is));

    } catch (MalformedURLException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

/*Genero i metodi getter e setter per l'accesso alla variabili private url e
content*/
public String getUrl() {
    return url;
}

public void setUrl(String url) {
    this.url = url;
}

public String getContent() {
    return content;
}

public void setContent(String content) {
    this.content = content;
}
}

class Utils {
    public static String getString(InputStream is) {
        BufferedReader br = null;
        StringBuilder sb = new StringBuilder();

        String line;
        try {
            br = new BufferedReader(new InputStreamReader(is));
            while ((line = br.readLine()) != null) {
                sb.append(line);
            }
        } catch (IOException e) {

```

```

        e.printStackTrace();
    } finally {
        if (br != null) {
            try {
                br.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}

return sb.toString();
}
}

class MainConcorrenza{
    public static void main(String[] args) {
        GetPaginaSito s1 = new GetPaginaSito("http://www.paolopreite.it");
        GetPaginaSito s2 = new GetPaginaSito("http://www.google.it");

        s1.start();
        s2.start();

/*Attendo il completamento di ciascun thread, prima di eseguire
quello successivo, in modo da non farli interrompere a vicenda*/
try {
    s1.join();
    s2.join();
} catch (InterruptedException e) {
    e.printStackTrace();
}

System.out.println("Output sito Paolo Preite");
System.out.println(s1.getContent());

System.out.println("Output sito Google");
System.out.println(s2.getContent());
}
}

```

CONCORRENZA CON L'UTILIZZO DEGLI EXECUTOR

Vediamo adesso come gestire la concorrenza attraverso gli Executor, che sono disponibili a partire dalla versione 1.5 di JAVA.

Dobbiamo creare innanzitutto una classe che implementi l'interfaccia Callable e che definisca il metodo call(). A questo punto, per eseguire i thread, dobbiamo creare un'istanza della classe ExecutorService ed utilizzare il metodo invokeAll(). Questo metodo ritorna una lista di oggetti Future, attraverso cui è possibile recuperare il valore di ritorno di ogni invocazione.

Vediamo di nuovo l'esempio della pagina web, ma con l'utilizzo degli Executor:

```
import java.io.IOException;
import java.io.InputStream;
import java.net.MalformedURLException;
import java.net.URL;
import java.net.URLConnection;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.*;

public class GetPaginaSitoExecutor implements Callable <String>{ /*String è il tipo di output*/
    private String url;
    private String content;

    public GetPaginaSitoExecutor(String url){
        super();
        this.url = url;
    }

    @Override
    public String call() throws Exception {

        try {
            URL u = new URL(url);

            URLConnection con = u.openConnection();

            InputStream is = con.getInputStream();

            return Utils.getString(is);

        } catch (MalformedURLException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
        return null;
    }

    public String getUrl() {
        return url;
    }

    public void setUrl(String url) {
        this.url = url;
    }
}
```

```

public String getContent() {
    return content;
}

public void setContent(String content) {
    this.content = content;
}
}

class MainConcorrenzaExecutor{
    public static void main(String[] args) throws InterruptedException,
ExecutionException {
        List<Callable<String>> siti = new ArrayList<Callable<String>>();

        siti.add(new GetPaginaSitoExecutor("http://www.paolopreite.it"));
        siti.add(new GetPaginaSitoExecutor("http://www.google.it"));

        ExecutorService ex = Executors.newSingleThreadExecutor();

        List<Future<String>> out = ex.invokeAll(siti);

        for(Future<String> future : out){ /*for each*/
            System.out.println(future.get());
        }

        ex.shutdown();/*Termino l'istanza dell'ExecutorService*/
    }
}

```

CONCORRENZA CON L'UTILIZZO DEL FRAMEWORK FORK/JOIN

Vediamo quest'ultimo in cui gestiamo la concorrenza tra thread attraverso il framework Fork/Join, disponibile dalla versione 1.7 di JAVA. Questo framework è una specializzazione del framework Executor.

In questo caso dobbiamo creare una classe che estenda la classe RecursiveTask ed implementi il metodo compute().

Per eseguire i thread dobbiamo creare un'istanza della classe ForkJoinPool ed utilizzare il metodo invoke().

Vediamo di nuovo l'esempio della pagina web, ma con l'utilizzo del framework Fork/Join:

```
import java.io.IOException;
import java.io.InputStream;
import java.net.MalformedURLException;
import java.net.URL;
import java.netURLConnection;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;

public class GetPaginaSitoForkJoin extends RecursiveTask<String> {
    private String url;
    private String content;

    public GetPaginaSitoForkJoin(String url){
        super();
        this.url = url;
    }

    @Override
    protected String compute() {
        try {
            URL u = new URL(url);

            URLConnection con = u.openConnection();

            InputStream is = con.getInputStream();

            return Utils.getString(is);

        } catch (MalformedURLException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
        return null;
    }

    public String getUrl() {
        return url;
    }

    public void setUrl(String url) {
        this.url = url;
    }

    public String getContent() {
        return content;
    }
}
```

```
public void setContent(String content) {
    this.content = content;
}
}

class MainConcorrenzaForkJoin{
    public static void main(String[] args) {
        ForkJoinPool f = new ForkJoinPool();

        System.out.println(f.invoke(new
GetPaginaSitoForkJoin("http://www.paolopreite.it")));
        System.out.println(f.invoke(new
GetPaginaSitoForkJoin("http://www.google.it")));

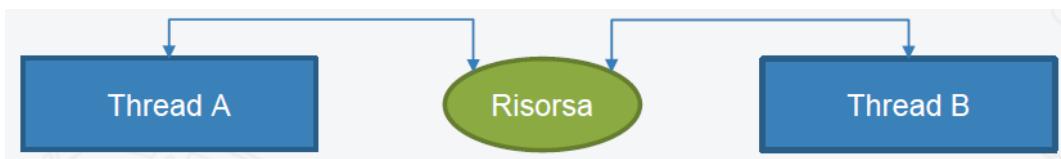
        f.shutdown();
    }
}
```

SINCRONIZZAZIONE

Per capire cosa è la sincronizzazione, vediamo un esempio. Supponiamo che Mario e Lucia abbiano un conto corrente bancario cointestato, quindi entrambi gli utenti possono accedere allo stesso conto corrente, hanno due bancomat e possono entrambi prelevare da due terminali diversi. Dobbiamo immaginare lo scenario in cui Mario e Lucia effettuano nello stesso istante la visualizzazione dell'estratto conto, vedono che il saldo è sufficiente ed effettuano il prelievo da due terminali diversi. Se non gestiamo il fatto che i due utenti accedono allo stesso conto corrente, Mario e Lucia possono effettuare il prelievo nello stesso istante, correndo il rischio che il saldo diventi negativo.

Mario e Lucia sono due thread che accedono alla stessa risorsa, ossia il conto corrente. Per poter prevenire eventuali situazioni anomale (ad esempio un saldo negativo sul conto corrente), è necessario gestire la sincronizzazione tra due thread che accedono alla stessa risorsa.

JAVA consente di gestire la sincronizzazione tra thread attraverso la keyword synchronized. Questa keyword consente di bloccare una risorsa, garantendo l'accesso esclusivo ad un thread. Quando questa risorsa è bloccata, nessun thread può accedervi finché la risorsa che l'ha bloccata non la libera.



Nell'esempio precedente, se Mario blocca la risorsa conto corrente durante il prelievo, Lucia non potrà prelevare finché Mario non avrà effettuato il prelievo. A quel punto verrà effettuato il controllo sul saldo e, quindi, se il saldo è disponibile, verrà effettuato il prelievo, altrimenti no.

Per quanto riguarda la keyword synchronized:

- in una classe possiamo definire più metodi synchronized, a seconda delle necessità;
- un metodo synchronized può essere eseguito solo da un thread alla volta;
- quando esistono più metodi synchronized in una classe, solo un metodo per volta può essere invocato;
- quando viene invocato un metodo synchronized, ovviamente il thread chiamante tecnicamente si dice che ottiene il lock, quindi blocca l'accesso da parte di altri thread a quella risorsa;
- i thread che vogliono accedere ad una risorsa bloccata rimangono in attesa di ricevere il lock, quindi rimangono in stato sospeso finché la risorsa non viene sbloccata dal thread che l'aveva occupata;
- quando un thread termina l'esecuzione di un metodo synchronized, il lock viene rilasciato e il metodo torna disponibile anche agli altri thread.

Vediamo l'esempio di Mario e Lucia con il codice:

Cliente.java

```
public class Cliente extends Thread {  
    private double sommaDaPrelevare;  
  
    public Cliente(String nomeCliente, double sommaDaPrelevare) {
```

```

        super();
        this.setName(nomeCliente);
        this.sommaDaPrelevare = sommaDaPrelevare;
    }

    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + " arriva al
bancomat");
        System.out.println("Quando arriva " +
Thread.currentThread().getName() + " il saldo è: " +
ContoCorrente.getInstance().getSaldo());
        System.out.println("La somma che vuole prelevare " +
Thread.currentThread().getName() + " è: " + sommaDaPrelevare);

        try {
            ContoCorrente.getInstance().prelievo(sommaDaPrelevare);
            System.out.println(Thread.currentThread().getName() + " "
TUTTO OK PRELIEVO EFFETTUATO");
        } catch (Exception e) {
            System.out.println(Thread.currentThread().getName() + " "
NOOOOOOOOO NON HAI SOLDI!!!!");
            e.printStackTrace();
        }
    }
}

```

ClienteNonSync.java

```

public class ClienteNonSync extends Thread {
    private double sommaDaPrelevare;

    public ClienteNonSync(String nomeCliente, double sommaDaPrelevare) {
        super();
        this.setName(nomeCliente);
        this.sommaDaPrelevare = sommaDaPrelevare;
    }

    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + " arriva al
bancomat");
        System.out.println("Quando arriva " +
Thread.currentThread().getName() + " il saldo è: " +
ContoCorrente.getInstance().getSaldo());
        System.out.println("La somma che vuole prelevare " +
Thread.currentThread().getName() + " è: " + sommaDaPrelevare);
        try {

            ContoCorrente.getInstance().prelievoNonSync(sommaDaPrelevare);
            System.out.println(Thread.currentThread().getName() + " "
TUTTO OK PRELIEVO EFFETTUATO");
        } catch (Exception e) {
            System.out.println(Thread.currentThread().getName() + " "
NOOOOOOOOO NON HAI SOLDI!!!!");
            e.printStackTrace();
        }
    }
}

```

ContoCorrente.java

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class ContoCorrente {
    private static ContoCorrente cc;

    public static ContoCorrente getInstance() {
        if(cc == null)
            cc = new ContoCorrente();

        return cc;
    }

    public double getSaldo() {
        double saldo = 0;

        BufferedReader br = null;
        try {
            File fin = new File(new File(".").getCanonicalPath() +
File.separator + "db.txt");

            br = new BufferedReader(new FileReader(fin));

            String line = null;
            while ((line = br.readLine()) != null) {
                saldo = Double.parseDouble(line);
                break;
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if(br != null)
                try {
                    br.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
        }
        return saldo;
    }

    public synchronized void prelievo(double somma) throws Exception {
        Thread.sleep(5000);

        BufferedWriter bw = null;
        FileWriter fw = null;

        try {
            double nuovoSaldo = getSaldo() - somma;

            if(nuovoSaldo > 0) {
                fw = new FileWriter(new
```

```

File(".").getCanonicalPath() + File.separator + "db.txt");
                                bw = new BufferedWriter(fw);
                                bw.write(nuovoSaldo+"");
                            } else
                                throw new Exception("Saldo insufficiente!");
} catch (IOException e) {
    e.printStackTrace();
} finally {
    try {
        if (bw != null)
            bw.close();

        if (fw != null)
            fw.close();
    } catch (IOException ex) {
        ex.printStackTrace();
    }
}

}
}

public void prelievoNonSync(double somma) throws Exception {
    Thread.sleep(5000);

    BufferedWriter bw = null;
    FileWriter fw = null;

    try {
        double nuovoSaldo = getSaldo() - somma;

        if(nuovoSaldo > 0) {
            fw = new FileWriter(new
File(".").getCanonicalPath() + File.separator + "db.txt");
            bw = new BufferedWriter(fw);
            bw.write(nuovoSaldo+"");
        } else
            throw new Exception("Saldo insufficiente!");
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        try {
            if (bw != null)
                bw.close();

            if (fw != null)
                fw.close();
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }

}
}
}

```

Main.java

```
public class Main {  
  
    public static void main(String[] args) throws InterruptedException {  
        Cliente c1 = new Cliente("Mario", 20);  
        Cliente c2 = new Cliente("Lucia", 50);  
  
        // Avvio i Threads  
        c1.start();  
        c2.start();  
  
        // Attendo il completamento  
        c1.join();  
        c2.join();  
  
        /*****  
  
        //  
        //  
        //  
        // Avvio i Threads  
        //  
        //  
        //  
        //  
        // Attendo il completamento  
        //  
        //  
        //  
        //  
    }  
}
```

La classe Cliente e la classe ClienteNonSync sono entrambi thread, perché estendono entrambe la classe Thread. All'interno di entrambe abbiamo la variabile sommaDaPrelevare e il costruttore che prende in ingresso il nome del cliente e la somma da prelevare. Importante da ricordare è che, quando creiamo un thread che estende la classe Thread, per passare delle variabili come parametri in ingresso, è necessario dichiarare tali variabili all'interno della classe ed implementare il costruttore. All'interno del metodo run() di entrambi i thread, abbiamo delle istruzioni che ci dicono chi arriva al bancomat, quanto è il saldo nel momento in cui l'utente arriva al bancomat e quanto è la somma da prelevare. Dopodichè, sempre nel metodo run(), abbiamo il blocco try catch, dove nel try vengono prelevati rispettivamente i metodi prelievoNonSync per la classe ClienteNonSync e prelievo per la classe Cliente. A questi due metodi viene passato in ingresso ovviamente la somma da prelevare. Se il prelievo viene effettuato, abbiamo il messaggio di "prelievo effettuato", altrimenti abbiamo un messaggio di "saldo negativo".

La classe ContoCorrente utilizza il singleton pattern e ha tre metodi, ossia getSaldo() che ritorna il saldo, prelievo() e prelievoNonSync(). Il codice tra gli ultimi due metodi è pressoché identico, cambia solo l'utilizzo della keyword synchronized per il metodo prelievo(). In entrambi i casi, se il prelievo viene effettuato con successo, viene riscritto il file db.txt con il nuovo saldo, altrimenti abbiamo un'eccezione.

Nella classe Main ci sono rispettivamente due istanze dei thread Cliente e ClienteNonSync, vengono avviati con il metodo start() e si attende il loro completamento con il metodo join().

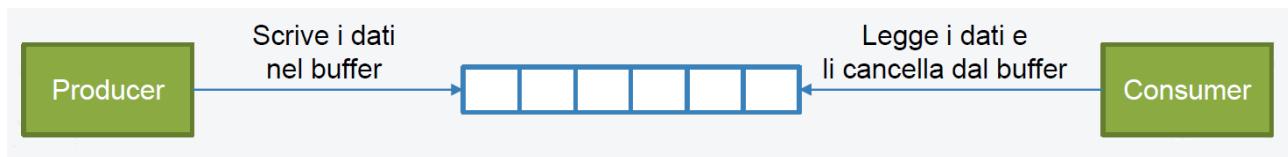
Con questo codice possiamo studiare il comportamento di due thread, sia quando abbiamo la

sincronizzazione, sia quando non ce l'abbiamo. Quando abbiamo la sincronizzazione, solo uno dei due thread entrerà nel metodo prelievo(), mentre l'altro resta in attesa. Quando non abbiamo la sincronizzazione, tutti e due i thread invece avranno libero accesso contemporaneamente al metodo prelievoNonSync()

ESEMPIO MULTITHREADING E CONCORRENZA : PRODUCER-CONSUMER

Il problema del producer-consumer (noto anche come problema del buffer limitato) è un classico esempio di sincronizzazione tra processi.

Ci sono i processi, producer e consumer, che condividono un buffer di dimensione fissa. Il buffer è una zona di memoria usata per compensare differenze di velocità nel trasferimento o nella trasmissione di dati, oppure per velocizzare l'esecuzione di alcune operazioni. Il producer genera dei dati e li scrive nel buffer, mentre il consumer legge i dati scritti dal producer e li cancella dal buffer. Quindi, da un lato abbiamo un utente che scrive, mentre dall'altro abbiamo un utente che legge e cancella.



Il problema consiste nell'assicurarsi che:

- il producer non elabori nuovi dati quando il buffer è pieno, ma che si metta in pausa;
- il consumer non cerchi di leggere dati quando il buffer è vuoto, ma che si metta in pausa.

La soluzione consiste nel:

- sospendere l'esecuzione del producer se il buffer è pieno. Quando il consumer preleva un elemento dal buffer, esso provvederà a svegliare il producer, che riprenderà a riempire il buffer;
- sospendere l'esecuzione del consumer se il buffer è vuoto. Quando il producer avrà inserito i dati nel buffer, esso provvederà a svegliare il consumer, che riprenderà a leggere e svuotare il buffer.

Questa soluzione può essere implementata mediante l'utilizzo dei semafori, che sono strategie di comunicazione tra processi. In questo caso bisogna fare molta attenzione perché, se non implementata correttamente, la soluzione porta ad avere una situazione di deadlock, in cui tutti e due i processi restano in attesa di essere risvegliati e non vengono risvegliati mai. Vediamo un esempio:

Consumer.java

```
import java.util.List;

public class Consumer implements Runnable {
    private final List<Integer> bufferCondiviso;

    public Consumer(List<Integer> bufferCondiviso, int size) {
        this.bufferCondiviso = bufferCondiviso;
    }
```

```

@Override
public void run() {
    while (true) {
        try {
            System.out.println("Il thread Consumer sta leggendo il buffer... ");
            consume();
            Thread.sleep(1000);
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
    }
}

private void consume() throws InterruptedException {
    // il thread resta in stato wait se il buffer è vuoto

    while (bufferCondiviso.isEmpty()) {
        synchronized (bufferCondiviso) {
            System.out.println("Il buffer è vuoto, il thread Consumer resta in
attesa... la dimensione del buffer adesso è: " + bufferCondiviso.size());

            bufferCondiviso.wait();
        }
    }

    // il buffer contiene elementi, quindi il thread può eliminarne uno e
    // notificarlo al producer
    synchronized (bufferCondiviso) {
        System.out.println("Il thread Consumer sta leggendo il buffer ed eliminando
il seguente elemento: " + bufferCondiviso.remove(0) + " la dimensione del buffer adesso
è: " + bufferCondiviso.size());

        bufferCondiviso.notifyAll();
    }
}
}

```

Producer.java

```
import java.util.List;

public class Producer implements Runnable {
    private final List<Integer> bufferCondiviso;
    private final int SIZE;
    private int i = 1;

    public Producer(List<Integer> bufferCondiviso, int size) {
        this.bufferCondiviso = bufferCondiviso;
        this.SIZE = size;
    }

    @Override
    public void run() {
        while(true) {
            try {
                produce();
                i++;
                Thread.sleep(1000);
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
        }
    }

    private void produce() throws InterruptedException {
        // il thread resta in stato wait se il buffer è pieno
        while (bufferCondiviso.size() == SIZE) {
            synchronized (bufferCondiviso) {
                System.out.println("Il buffer è pieno, il
thread Producer resta in attesa... la dimensione del buffer adesso è: " +
bufferCondiviso.size());
                bufferCondiviso.wait();
            }
        }

        // il buffer non è pieno, quindi il thread può aggiungere un
nuovo elemento e notificarlo al consumer
        synchronized (bufferCondiviso) {
            bufferCondiviso.add(i);
            bufferCondiviso.notifyAll();

            System.out.println("Il thread Producer ha aggiunto al
buffer l'elemento: " + i + " la dimensione del buffer adesso è: " +
bufferCondiviso.size());
        }
    }
}
```

ProducerConsumerTest.java

```
import java.util.LinkedList;
import java.util.List;

public class ProducerConsumerTest {
    public static void main(String args[]) {
        List<Integer> bufferCondiviso = new LinkedList<Integer>();
        int size = 4;

        Thread prodThread = new Thread(new Producer(bufferCondiviso, size),
"Producer");
        Thread consThread = new Thread(new Consumer(bufferCondiviso, size),
"Consumer");

        prodThread.start();
        consThread.start();
    }
}
```

Abbiamo le classi Producer e Consumer che sono due thread e, infatti, implementano Runnable. Abbiamo in entrambe le classi la lista bufferCondiviso, che è condivisa tra le classi stesse.

Nel caso del Consumer, nel metodo run() viene invocato il metodo consume(). Nel metodo consume(), se il bufferCondiviso è vuoto, viene invocato il metodo synchronized con parametro il bufferCondiviso e il thread Consumer rimane in attesa (con l'utilizzo del metodo wait()) che il bufferCondiviso sia di nuovo disponibile con degli elementi. Se il bufferCondiviso, invece, contiene gli elementi, il thread Consumer può rimuovere il primo elemento della lista (bufferCondiviso.remove(0)) e notifica (bufferCondiviso.notifyAll()) al Producer, che è in attesa sulla variabile bufferCondiviso, che potrà continuare a inserire dati.

Nel caso del Producer, nel metodo run() viene invocato il metodo produce(). Nel metodo produce(), se il bufferCondiviso è pieno, viene invocato il metodo synchronized con parametro il bufferCondiviso e il thread Producer rimane in attesa (con l'utilizzo del metodo wait()) che il bufferCondiviso sia di nuovo disponibile. Se il bufferCondiviso, invece, è vuoto, il thread Producer può aggiungere elementi dalla lista (bufferCondiviso.add(i)) e notifica (bufferCondiviso.notifyAll()) al Consumer, che è in attesa sulla variabile bufferCondiviso, che il buffer contiene degli elementi.

Quindi, attraverso l'implementazione dell'esempio Producer-Consumer, due thread che accedono allo stesso buffer (alla stessa risorsa), rispettivamente in lettura dal lato Consumer e in scrittura dal lato Producer, abbiamo visto come implementare il multithreading e la concorrenza in JAVA. Come abbiamo notato, attraverso la parola chiave synchronized, abbiamo bloccato l'accesso ad una variabile (in questo caso bufferCondiviso). Quando il Producer effettua il lock sulla variabile, il Consumer resta in attesa che venga liberato il lock, quindi che la variabile venga resa disponibile e viceversa.

METODI WAIT(), NOTIFY(), NOTIFYALL()

I metodi `wait()`, `notify()` e `notifyAll()` sono definiti all'interno della classe `Object`. La classe `Object` è la superclasse da cui derivano tutte le altre classi di JAVA.

WAIT()

Il metodo `wait()`:

- mette in attesa un thread;
- possiamo invocarlo solo su oggetti per il quale si ha il lock;
- possiamo invocarlo solo in un metodo o in un blocco di codice `synchronized`, altrimenti avremo l'eccezione `IllegalMonitorStateException`.

Quando viene invocato il metodo `wait()` su un oggetto, si hanno i seguenti effetti:

- sull'oggetto viene rilasciato il lock;
- il thread viene posto in stato `blocked`.

Analogamente al metodo `wait()`, anche `sleep()` mette in attesa il thread invocante. Tra i due, però, c'è una differenza:

- quando viene invocato il metodo `sleep()`, il lock sull'oggetto non viene rilasciato, quindi nessun thread può utilizzarlo;
- quando viene invocato il metodo `wait()`, invece, viene rilasciato il lock sull'oggetto, il quale diventa accessibile agli altri thread.

Esistono diverse implementazioni del metodo `wait()`:

- `wait()` causa l'interruzione di un thread finché un altro thread non invoca il metodo `notify()` o `notifyAll()`;
- `wait(long timeout)` causa l'interruzione di un thread finché un altro thread non invoca il metodo `notify()`, `notifyAll()`, o se è stato raggiunto il timeout, espresso in millisecondi, impostato. Se il timeout passato in ingresso è 0, il comportamento è lo stesso del metodo `wait()`;
- `wait(long timeout, int nanos)` è analogo a `wait (long timeout)`, solo che al timeout in millisecondi è possibile aggiungere anche i nanosecondi.

NOTIFY() E NOTIFYALL()

Il metodo notify() risveglia un solo thread in attesa su un oggetto che si trovava in stato di lock.

Il metodo notifyAll() risveglia tutti i thread in attesa su un oggetto che si trovava in stato di lock.

Quando sono invocati questi due metodi, i thread che ricevono la notifica passano nello stato Runnable, quindi possono da quel momento in poi prendere il lock sulla risorsa. Per poter accedere o effettuare modifiche su una risorsa synchronized, i thread risvegliati devono acquisire il lock.

Un thread può invocare i due metodi notify() e notifyAll() solo se ha il lock sulla risorsa.

Quando viene invocato il metodo notifyAll() abbiamo che:

- tutti i thread in attesa vengono risvegliati;
- quando vengono risvegliati, tutti quei thread si mettono in coda e il primo che riesce ad acquisire la risorsa, effettua il lock;
- solo un thread per volta può prendere il lock, mentre gli altri dovranno di nuovo attendere che la risorsa venga rilasciata.

SINCRONIZZAZIONE AVANZATA CON LOCK E REENTRANTLOCK

Come abbiamo già visto, possiamo sincronizzare un blocco di codice, un metodo o una variabile attraverso la keyword synchronized. L'utilizzo di questa keyword consente di accedere in maniera esclusiva alla risorsa, impedendo che altri thread possano accedere al stessa risorsa mentre è utilizzata.

L'utilizzo della keyword synchronized, tuttavia, ha delle limitazioni, ovvero:

- se abbiamo in una classe più metodi synchronized, possiamo effettuare il lock di una risorsa per volta;
- quando un thread effettua il lock su un metodo, non può effettuare l'unlock finché non ha eseguito tutto il blocco di codice;
- non possiamo effettuare l'esecuzione di un blocco di codice sincronizzato.

Attraverso l'interfaccia Lock, queste limitazioni vengono superate.

LOCK

L'Interfaccia Lock, appunto, è stata pensata per offrire gli elementi messi a disposizione dalla keyword synchronized, più altri elementi che consentono di superare le limitazioni di synchronized.

I metodi più importanti di questa interfaccia sono:

- lock(), che effettua il lock di una risorsa;
- unlock(), che libera una risorsa;
- tryLock(), che attende un certo periodo di tempo, prima di effettuare il lock.

REENTRANTLOCK

La classe ReentrantLock è un'implementazione dell'interfaccia Lock ed è disponibile dalla versione 1.5 di JAVA. Questa classe, oltre ad implementare i metodi dell'interfaccia Lock, contiene altri metodi utili.

Quando utilizziamo la classe ReentrantLock, il lock è rientrante. Questo vuol dire che un thread che ha un lock, può acquisire nuovamente il lock più volte.

Vediamo un esempio con il metodo lock():

```
public class ReentrantLockEsempio {  
    private ReentrantLock lock = new ReentrantLock();  
    private int contatore = 0;  
  
    public int conta() {  
        lock.lock();  
  
        try {  
            System.out.println(Thread.currentThread().getName() + " contatore = " + contatore);  
            contatore++;  
  
            return contatore;  
        } finally {  
            lock.unlock();  
        }  
    }  
}
```

Abbiamo una classe ReentrantLockEsempio, che ha una variabile lock e un metodo conta(). Tutto il codice che si trova dopo l'invocazione del metodo lock, deve essere inserita all'interno di un blocco try. All'interno del blocco finally, invece, bisogna inserire l'unlock(). Quando un thread acquisisce il lock sull'istanza della classe ReentrantLock (variabile lock), nessun altro thread può andare oltre la riga di codice in cui abbiamo la chiamata al metodo lock(). Quindi se un altro thread richiede il lock, rimane bloccato sulla riga di codice in cui viene richiamato il metodo lock(), finché il lock non viene sbloccato.

Vediamo un esempio con il metodo tryLock():

```
public class ReentrantLockEsempio {  
    private ReentrantLock lock = new ReentrantLock();  
    private int contatore = 0;  
  
    public void somma() {  
        System.out.println("Il thread " + Thread.currentThread().getName() + " ha richiesto ...");  
  
        if(lock.tryLock()) {  
            try {  
                somma += contatore;  
                System.out.println(Thread.currentThread().getName() + " la somma vale = " + somma);  
            } finally {  
                lock.unlock();  
            }  
        } else {  
            .....  
        }  
    }  
}
```

In questo esempio abbiamo la stessa classe di prima, in cui però stavolta abbiamo il metodo tryLock(). Attraverso questo metodo, un thread può tentare di acquisire il lock sull'istanza della classe ReentrantLock (variabile lock), senza rimanere in sospeso come nell'esempio precedente a questo.

Esiste anche un altro metodo tryLock(), che prende in ingresso il parametro timeout che specifica il tempo di attesa prima di effettuare un altro tentativo di lock, e lockunit che specifica l'unità di tempo relativa al timeout(secondi, millisecondi...).

UTILIZZARE IL BLOCCO TRY - FINALLY CON I THREAD

Il problema consiste nell'avere un thread, al suo interno abbiamo un blocco try/catch/finally e il thread viene interrotto durante la sua esecuzione. Si deve capire cosa succede quando avviene questa interruzione.

Partiamo con il distinguere due tipi di interruzione:

- se un thread sta eseguendo le istruzioni all'interno del try o del catch e viene terminato (cioè killed), il blocco finally potrebbe non essere eseguito. Questo è il caso in cui il processo, associato alla JVM, muore durante l'esecuzione del thread;
- se un thread sta eseguendo le istruzioni all'interno del try o del catch e viene interrotto(tremite il metodo interrupt()), il blocco finally viene eseguito.

Vediamo un esempio:

EsempioTryCatchFinally.java

```
public class EsempioTryCatchFinally extends Thread {  
    private long sleep;  
  
    public EsempioTryCatchFinally(long sleep) {  
        super();  
        this.sleep = sleep;  
    }  
  
    @Override  
    public void run() {  
        try {  
            Thread.sleep(sleep);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        } finally {  
            System.out.println("Entrato nel finally!");  
        }  
    }  
}
```

Main2.java

```
public class Main2 {  
  
    public static void main(String[] args) throws InterruptedException {  
        EsempioTryCatchFinally t1 = new EsempioTryCatchFinally(10);  
  
        // Avvio il Thread  
        t1.start();  
  
        // Interromo il thread  
        t1.interrupt();  
    }  
  
}
```

Qui abbiamo una classe EsempioTryCatchFinally, che estende la classe Thread, in cui abbiamo un blocco try/catch/finally. Nella classe Main2 creo un'istanza della classe EsempioTryCatchFinally, avvio il thread e lo interromo con il metodo interrupt(). L'output che ci esce fuori ci avvisa che è riuscito ad entrare nel blocco finally.

THREAD POOL

I thread pool sono dei componenti software che si occupano di gestire i thread, con l'obiettivo di ottimizzare e semplificare l'utilizzo.

Quando scriviamo un software multi-threading, abbiamo visto che ogni thread ha il compito di eseguire un determinato task. Attraverso il thread pool possiamo gestire l'esecuzione di una lista di thread.

Ovviamente il thread pool è dotato di una sua coda interna, che consente di aggiungere più thread, accodandoli tra loro. La gestione dell'esecuzione dei vari thread viene lasciata in carico al thread pool.

Per decidere quale thread deve essere eseguito per primo e quanti eseguirne, esistono diversi algoritmi. In JAVA sono state implementate diverse classi che implementano questi algoritmi e consentono di definire diversi thread pool, a seconda della necessità.

I motivi per cui utilizzare il thread pool sono i seguenti:

- abbiamo un aumento delle prestazioni delle applicazioni che li utilizzano, poiché il thread pool ottimizza l'utilizzo della RAM e della CPU;
- I task vengono eseguiti in maniera più veloce, perché le operazioni vengono parallelizzate, avendo un thread pool che consente di eseguire più thread contemporaneamente;
- dal punto di vista del codice sorgente, abbiamo una maggiore eleganza di scrittura. Quindi il codice risulta essere più pulito, perché non dobbiamo occuparci della creazione e della gestione dei thread, ovvero non dobbiamo fare più una cosa del genere:

```
public static void main(String[] args) {  
  
    EsempioMultithreading em1 = new EsempioMultithreading();  
    //setName() serve per poter assegnare un nome al thread  
    em1.setName("Thread1");  
  
    EsempioMultithreading em2 = new EsempioMultithreading();  
    em2.setName("Thread2");  
  
    EsempioMultithreading em3 = new EsempioMultithreading();  
    em3.setName("Thread3");  
  
    EsempioMultithreading em4 = new EsempioMultithreading();  
    em4.setName("Thread4");  
  
    EsempioMultithreading em5 = new EsempioMultithreading();  
    em5.setName("Thread5");  
  
    em1.start();  
    em2.start();  
    em3.start();  
    em4.start();  
    em5.start();  
  
}
```

Per implementare i thread pool in JAVA abbiamo l'interfaccia Executor, l'interfaccia ExecutorService e la classe Executors.

L'interfaccia Executor è l'interfaccia base che definisce quali sono i meccanismi principali per la gestione di

un thread pool. Ovviamente, essendo un' interfaccia, definisce dei metodi, ma non li implementa. Il metodo principale execute(thread da eseguire) consente di aggiungere nuovi thread al pool.

L'interfaccia ExecutorService estende l'interfaccia Executor ed aggiunge una serie di metodi per ottimizzare le gestione del pool, tra cui shutdown() che indica al pool di avviare la chiusura di tutti i thread. Dopo il metodo shutdown(), non posso aggiungere più nuovi thread al pool.

La classe Executors è una classe factory che consente di creare varie istanze di pool (Executor, ExecutorService,...).

Facciamo un esempio, riprendendo il caso in cui dobbiamo ottenere l'output in html di una pagina web:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.URL;
import java.netURLConnection;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class GetPaginaSitoPool extends Thread {
    private String url;
    private String content;

    public GetPaginaSitoPool(String url) {
        super();
        this.url = url;
    }

    @Override
    public void run() {
        try {
            URL site = new URL(url);
            URLConnection con = site.openConnection();

            InputStream in = con.getInputStream();
            String encoding = con.getContentEncoding();
            encoding = encoding == null ? "UTF-8" : encoding;

            System.out.println("*****");
            System.out.println("CONTENUTO DELLA PAGINA WEB: " + url);
            System.out.println(getString(in));
            System.out.println("*****");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private String getString(InputStream is) {
        BufferedReader br = null;
        StringBuilder sb = new StringBuilder();

        String line;
        try {
            br = new BufferedReader(new InputStreamReader(is));
            while ((line = br.readLine()) != null) {
                sb.append(line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (br != null) {
                try {
                    br.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
        return sb.toString();
    }
}
```

```

        }

    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        if (br != null) {
            try {
                br.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }

    return sb.toString();
}

public String getContent() {
    return content;
}
}

class MainPool{
    public static void main(String[] args) {
        /*creo il thread pool*/
        ExecutorService pool = Executors.newCachedThreadPool();

        /*aggiunge i thread al pool*/
        pool.execute(new GetPaginaSitoPool("http://www.google.com"));
        pool.execute(new GetPaginaSitoPool("https://www.ibconline.it"));

        /*spengo il thread pool*/
        pool.shutdown();
    }
}

```

Nella classe MainPool creiamo un'istanza di ExecutorService, che crea un thread pool. All'interno del pool aggiungiamo, attraverso il metodo execute(), i thread che ci interessano. In alternativa al CachedThreadPool(), che è un pool in cui potenzialmente possiamo inserire infiniti thread, abbiamo il newFixedThreadPool(int n Threads), in cui il numero di thread che si possono inserire in questo pool è dato da quello specificato dal parametro in ingresso n Threads. Sempre nel caso del newFixedThreadPool(int n Threads), se ci sono più thread di quelli ammissibili, i restanti verranno accodati in attesa di essere eseguiti.

È caldamente consigliato di utilizzare sempre i thread pool quando lavoriamo in ambiente multithreading, perché consentono di effettuare dei lavori sui thread in maniera molto più semplice rispetto ai thread classici delle prime versioni di JAVA.

CLASSI ARRAYBLOCKINGQUEUE E LINKEDBLOCKINGQUEUE

Prima di parlare di queste due classi, vediamo quali sono le interfacce messe a disposizione da JAVA per rappresentare una generica coda. Le due classi ArrayBlockingQueue e LinkedBlockingQueue le utilizziamo, appunto, per la gestione delle code di thread.

In JAVA, una generica coda è rappresentata dall'interfaccia Queue, che estende l'interfaccia Collection. Estendendo l'interfaccia Collection, ne eredita tutte le caratteristiche (ad esempio i metodi add(e), remove(e), size() e così via).

In aggiunta ai metodi definiti dall'interfaccia Collection, l'interfaccia Queue ne definisce altri, tra cui:

- peek(), che recupera il primo elemento della coda, senza eliminarlo. Questo metodo ritorna l'elemento recuperato, oppure null se la coda è vuota;
- element(), che recupera, come il metodo peek(), il primo elemento della coda, con la differenza che se la coda è vuota, genera l'eccezione NoSuchElementException al posto di null;
- poll(), che recupera e rimuove il primo elemento della coda. Questo metodo ritorna l'elemento rimosso, oppure null se la coda è vuota.

L'interfaccia Queue viene estesa dall'interfaccia BlockingQueue, che rappresenta anch'essa una generica coda bloccante, solo che quest'ultima definisce dei metodi che devono garantire l'esecuzione sicura delle operazioni. In particolare, tali metodi sono:

- put(), che inserisce un oggetto alla fine della coda. Se la coda è piena, il metodo si blocca e mette il thread corrente in attesa, riattivandolo quando viene rimosso un elemento dalla coda;
- take(), che restituisce il primo elemento della coda. Se la coda è vuota, il metodo si blocca e mette il thread corrente in attesa, riattivandolo quando viene inserito un nuovo elemento alla coda.

Chiaramente, definendo la possibilità di mettere in attesa i thread, i software che utilizzano questa interfaccia sono sincronizzati.

Badiamo bene che si parla di interfaccia e non di classe, quindi l'interfaccia BlockingQueue non implementa questa logica, ma la definisce. L'interfaccia BlockingQueue, infatti, è implementata da diverse classi. Le classi che la implementano sono Thread Safe e sono:

- ArrayBlockingQueue
- LinkedBlockingQueue

CLASSE ARRAYBLOCKINGQUEUE

La classe ArrayBlockingQueue è un array circolare di tipo bloccante. Un oggetto di tipo ArrayBlockingQueue ha la particolarità di avere una capacità fissa, definita in fase di inizializzazione. Quindi nel costruttore dobbiamo passare la capacità e quella rimarrà.

Gli elementi sono ordinati all'interno della coda secondo le specifiche FIFO (First-In First-Out), cioè il primo elemento in ingresso è il primo ad uscire.

Riprendiamo l'esempio del Produttore/Consumatore, sostituendo la variabile synchronized con un BlockingQueue:

Producer.java

```
import java.util.concurrent.BlockingQueue;

public class Producer implements Runnable {
    private BlockingQueue<String> queue;

    public Producer(BlockingQueue<String> queue) {
        this.queue = queue;
    }

    @Override
    public void run() {
        int i = 0;

        while(true) {
            String elem = "Elemento numero " + i;

            /* provo ad aggiungere un elemento alla coda */
            boolean aggiunto = queue.offer(elem);

            System.out.println("L'elemento " + i + " " + (aggiunto ? "è stato aggiunto" : "non è stato aggiunto"));
            i++;
        }
    }
}
```

Consumer.java

```
import java.util.concurrent.BlockingQueue;

public class Consumer implements Runnable {
    private BlockingQueue<String> queue;

    public Consumer(BlockingQueue<String> queue) {
        this.queue = queue;
    }
}
```

```

@Override
public void run() {

    while(true) {
        if (queue.remainingCapacity() > 0) {
            System.out.println("E' possibile aggiungere
ancora " + queue.remainingCapacity() + " su " + queue.size());
        } else if (queue.remainingCapacity() == 0) {
            String elementoRimosso = queue.remove();

            System.out.println("E' stato rimosso
l'elemento " + elementoRimosso);
        }

        try {
            Thread.sleep(500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

EsempioArrayBlockingQueue.java

```

import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;

public class EsempioArrayBlockingQueue {
    public static void main(String[] args) {
        // Creo una coda che può contenere al massimo 10 elementi.
        BlockingQueue<String> queue = new ArrayBlockingQueue<String>(10);

        // Producer e Consumer accedono alla stessa coda...
        Thread prod = new Thread(new Producer(queue));
        Thread cons = new Thread(new Consumer(queue));

        prod.start();
        cons.start();
    }
}

```

Quindi, nella classe Producer abbiamo una variabile queue, che è di tipo BlockingQueue. Questa variabile viene inizializzata nel costruttore. All'interno del metodo run() abbiamo un ciclo while(true), che è un loop infinito finché non interrompiamo il software. All'interno del loop, il thread Producer prova ad aggiungere alla coda un elemento di tipo stringa, attraverso l'invocazione del metodo offer() definito dall'interfaccia BlockingQueue.

Nella classe Consumer abbiamo, analogamente alla classe Producer, una variabile queue di tipo BlockingQueue. Questa variabile viene inizializzata nel costruttore. All'interno del metodo run() abbiamo un ciclo while(true), che è un loop infinito finché non interrompiamo il software. All'interno del loop, il thread Consumer legge la coda e, se è piena, rimuove il primo elemento aggiunto (logica FIFO).

Poi abbiamo la classe EsempioArrayBlockingQueue che crea una variabile queue di tipo BlockingQueue, utilizzando la classe ArrayBlockingQueue e passando in ingresso la capacità massima di 10 elementi.

Dopodichè vengono creati due thread, uno per il produttore e uno per il consumatore, e vengono avviati.

CLASSE LINKEDBLOCKINGQUEUE

La classe LinkedBlockingQueue, a differenza della classe ArrayBlockingQueue, consente di creare istanze senza specificare la capacità, cioè ha il costruttore che non prende in ingresso la capacità. In questo caso, la capacità massima sarà Integer.MAX_VALUE, ovvero $2^{31}-1$ elementi (2.147.483.647).

Chiaramente, se la coda non è mai piena, il metodo put() (o il metodo offer()), che inserisce elementi, non si può mai bloccare. Riprendiamo l'esempio precedente, ma sostituendo la classe EsempioArrayBlockingQueue con EsempioLinkedBlockingQueue

EsempioLinkedBlockingQueue.java

```
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;

public class EsempioLinkedBlockingQueue {
    public static void main(String[] args) {
        // Creo una coda senza specificare la capacità.
        BlockingQueue<String> queue = new LinkedBlockingQueue<String>();

        // Producer e Consumer accedono alla stessa coda...
        Thread prod = new Thread(new Producer(queue));
        Thread cons = new Thread(new Consumer(queue));

        prod.start();
        cons.start();
    }
}
```

In questo caso, il Consumer entrerà in gioco solo quando il Producer avrà inserito 2.147.483.647.

Notiamo che, in entrambi i casi, non abbiamo utilizzato la keyword synchronized, perché l'interfaccia BlockingQueue definisce dei metodi che sono già sincronizzati e sono implementati nelle classi ArrayBlockingQueue e LinkedBlockingQueue. Quindi, tali metodi consentono di gestire in automatico le code, senza l'utilizzo della keyword synchronized.

GESTIONE BANCONE SALUMI MEDIANTE THREAD POOL

Partiamo dallo scenario di riferimento:

Siamo i proprietari di un supermercato. Al bancone dei salumi e formaggi abbiamo 3 dipendenti. Nel supermercato ci sono 30 clienti che devono acquistare salumi e formaggi. Ovviamente i 30 clienti arrivano vicino al bancone e prendono dal distributore del ticket il proprio numero. Preso il numero, il cliente si mette in attesa di essere servito.

Dobbiamo considerare che:

- ogni cliente ha la propria lista della spesa, quindi ognuno di essi impiegherà un certo numero di minuti per essere servito, che può essere diverso dal tempo impiegato dagli altri clienti;
- tutti e 30 i clienti non possono essere serviti contemporaneamente, perché abbiamo solo 3 dipendenti;
- appena si libera un dipendente, il prossimo cliente viene servito.

Per la gestione del bancone dei salumi e formaggio in codice, utilizziamo ciò che segue:

- la classe `ArrayBlockingQueue`: utilizzata per creare l'oggetto che rappresenta la nostra coda al bancone dei salumi e formaggi;
- la classe `ExecutorService`: è la classe factory utilizzata per creare il thread pool. Avremo tanti thread per quanti sono i dipendenti disponibili al bancone;
- la classe `Cliente`: è la classe che rappresenta il generico cliente. Ogni cliente è un thread, pertanto questa classe implementa l'interfaccia `Runnable`.

Vediamo il codice per questo esempio:

Cliente.java

```
import java.util.Random;

public class Cliente implements Runnable {
    private int numeroTicket;

    public Cliente(int numeroTicket) {
        System.out.println("E' arrivato un nuovo cliente ed ha preso il numero " + numeroTicket);

        this.numeroTicket = numeroTicket;
    }

    public void run() {
        /* il cliente ordina i prodotti al dipendente presente al bancone */
        richiediProdotti();
    }

    private void richiediProdotti() {
        System.out.println("Viene servito il cliente numero " +
numeroTicket);

        /* imposta una durata random per ciascun cliente... */
        Random r = new Random();
    }
}
```

```

/* per semplicità ipotizzo che ogni cliente impieghi tra 5 e 20
secondi per acquistare salumi e formaggi */
int tempoImpiegatoPerAcquisto = (r.nextInt(15) + 5)*1000;

try {
    /*
     * il thread viene sospeso per tempoImpiegatoPerAcquisto
     * Quest'attesa equivale al cliente che sta effettuando
     L'ordine al dipendente del bancone
    */
    Thread.sleep(tempoImpiegatoPerAcquisto);
} catch (InterruptedException e) {
    e.printStackTrace();
}

System.out.println("Il cliente che aveva il numero " + numeroTicket
+ " ha completato il suo acquisto in " + tempoImpiegatoPerAcquisto/1000 + " secondi");
}
}

```

BanconeSalumeriaFormaggi.java

```

import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class BanconeSalumeriaFormaggi {
    private final static int CLIENTI_DA_SERVIRE = 30;
    private final int DIPENDENTI_AL_BANCONE = 3;

    private BlockingQueue<Runnable> codaBancone = new
ArrayBlockingQueue<Runnable>(30, true);
    private ExecutorService dipendentiDisponibili =
Executors.newFixedThreadPool(DIPENDENTI_AL_BANCONE);

    public static void main(String[] args) {
        System.out.println("Nel supermercato ci sono " + CLIENTI_DA_SERVIRE
+ " clienti che stanno andando al bancone");

        BanconeSalumeriaFormaggi bancone = new BanconeSalumeriaFormaggi();
        bancone.arrivoClientiAlBancone();
        bancone.servizioClienti();
    }

    private void arrivoClientiAlBancone() {
        for (int i = 1; i <= CLIENTI_DA_SERVIRE; i++) {
            try {
                /* il cliente viene inserito in coda */
                codaBancone.put(new Cliente(i));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

```

```

        private void servizioClienti() {
            new Thread(new Runnable() {
                public void run() {
                    while(true) {
                        try {
                            /* il primo cliente
disponibile viene servito ... */

                            dipendentiDisponibili.execute(codaBancone.take());
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                    }
                }).start();
        }
    }
}

```

La classe Cliente implementa Runnable e ha una sola variabile, cioè numeroTicket. Per ogni istanza di Cliente, assegniamo un ticket, simulando proprio lo scenario in cui il cliente arriva al bancone e prende il proprio numero. Nel metodo run() abbiamo la chiamata al metodo richiediProdotti(). Il metodo richiediProdotti stampa che viene servito il cliente, imposta un tempo random per ciascun cliente, suppone che ogni cliente impieghi tra i 5 e 20 secondi per acquistare e, infine, il thread viene sospeso per il tempo impiegato per l'acquisto, simulando l'acquisto al bancone. Dopo che viene effettuato l'acquisto, il thread si risveglia e viene stampata che “ il cliente che aveva il numero n ha completato il suo acquisto in m secondi”.

Nella classe BanconeSalumeriaFormaggi abbiamo il numero di clienti da servire (30), il numero di dipendenti al bancone(3), la coda ArrayBlockingQueue che ha capacità 30 e i dipendenti disponibili e un thread pool di dimensione 3. Abbiamo anche i due metodi arrivoClientiAlBancone() e servizioClienti(). Quando viene creata, nel metodo main(), l'istanza della classe BanconeSalumeriaFormaggi, il primo metodo che viene invocato è arrivoClientiAlBancone(), che è un metodo che itera da 1 a CLIENTI_DA_SERVIRE e aggiunge alla coda un'istanza della classe Cliente (un nuovo thread), mentre il secondo metodo che viene invocato è servizioClienti(), in cui abbiamo un thread che viene avviato e permette di servire il primo cliente disponibile.

L'output del codice è il seguente:

Nel supermercato ci sono 30 clienti che stanno andando al bancone
E' arrivato un nuovo cliente ed ha preso il numero 1
E' arrivato un nuovo cliente ed ha preso il numero 2
E' arrivato un nuovo cliente ed ha preso il numero 3
E' arrivato un nuovo cliente ed ha preso il numero 4
E' arrivato un nuovo cliente ed ha preso il numero 5
E' arrivato un nuovo cliente ed ha preso il numero 6
E' arrivato un nuovo cliente ed ha preso il numero 7
E' arrivato un nuovo cliente ed ha preso il numero 8
E' arrivato un nuovo cliente ed ha preso il numero 9
E' arrivato un nuovo cliente ed ha preso il numero 10
E' arrivato un nuovo cliente ed ha preso il numero 11
E' arrivato un nuovo cliente ed ha preso il numero 12
E' arrivato un nuovo cliente ed ha preso il numero 13
E' arrivato un nuovo cliente ed ha preso il numero 14
E' arrivato un nuovo cliente ed ha preso il numero 15

E' arrivato un nuovo cliente ed ha preso il numero 16
E' arrivato un nuovo cliente ed ha preso il numero 17
E' arrivato un nuovo cliente ed ha preso il numero 18
E' arrivato un nuovo cliente ed ha preso il numero 19
E' arrivato un nuovo cliente ed ha preso il numero 20
E' arrivato un nuovo cliente ed ha preso il numero 21
E' arrivato un nuovo cliente ed ha preso il numero 22
E' arrivato un nuovo cliente ed ha preso il numero 23
E' arrivato un nuovo cliente ed ha preso il numero 24
E' arrivato un nuovo cliente ed ha preso il numero 25
E' arrivato un nuovo cliente ed ha preso il numero 26
E' arrivato un nuovo cliente ed ha preso il numero 27
E' arrivato un nuovo cliente ed ha preso il numero 28
E' arrivato un nuovo cliente ed ha preso il numero 29
E' arrivato un nuovo cliente ed ha preso il numero 30
Viene servito il cliente numero 1
Viene servito il cliente numero 2
Viene servito il cliente numero 3
Il cliente che aveva il numero 1 ha completato il suo acquisto in 6 secondi
Viene servito il cliente numero 4
Il cliente che aveva il numero 3 ha completato il suo acquisto in 10 secondi
Viene servito il cliente numero 5
Il cliente che aveva il numero 4 ha completato il suo acquisto in 5 secondi
Viene servito il cliente numero 6
Il cliente che aveva il numero 2 ha completato il suo acquisto in 14 secondi
Viene servito il cliente numero 7
....

Da qui capiamo che i 30 clienti vengono aggiunti alla coda, dopodichè vediamo quale cliente viene servito e in quanti secondi.

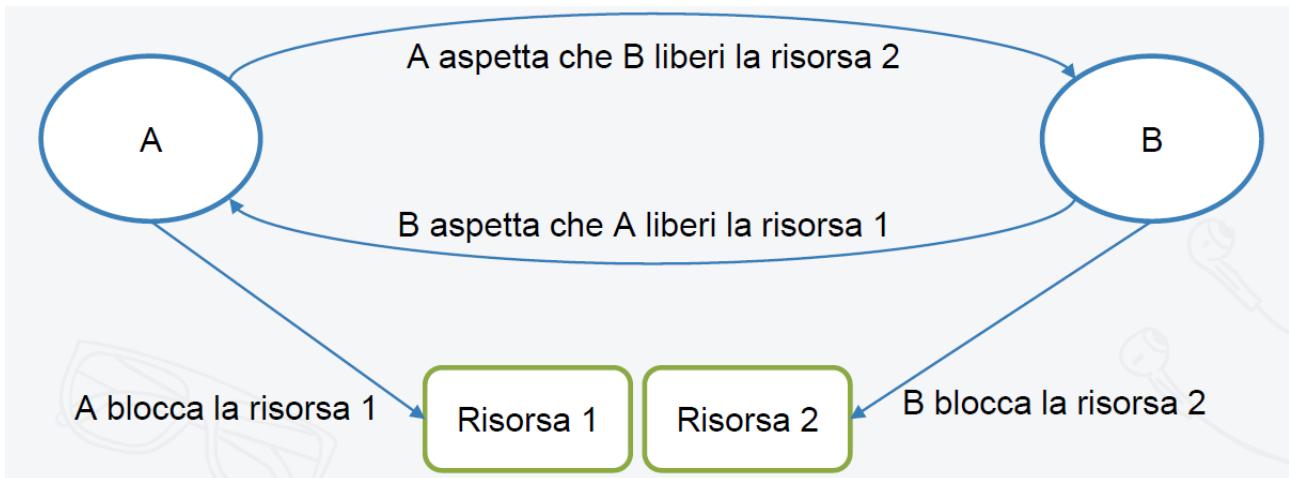
Riassumendo, in questo esempio abbiamo utilizzato la classe `ArrayBlockingQueue`, la coda `BlockingQueue` e il thread pool che indica il numero di thread che possiamo gestire simultaneamente. La coda è riempita da tutti i clienti. Un cliente è semplicemente un thread che effettua delle operazioni. Una volta serviti tutti e 30 i clienti, il software terminerà.

DEADLOCK, STARVATION E LIVELOCK

DEADLOCK

Il deadlock si ha quando un thread, che chiamiamo thread A, si blocca in attesa che il thread B liberi una risorsa condivisa tra i due e, a sua volta, il thread B resta bloccato in attesa che il thread A liberi un'altra risorsa. Quindi, il thread A aspetta che il thread B liberi una risorsa, mentre il thread B aspetta che il thread A liberi un'altra risorsa, bloccandosi a vicenda.

Il deadlock è uno di quei casi in cui non c'è via d'uscita.



STARVATION

La starvation si ha quando un thread non riesce mai ad acquisire le risorse di cui necessita , oppure ci riesce dopo troppo tempo, perché sono bloccate da altri thread.

Ad esempio, supponiamo di avere 3 thread A, B, C che accedono alla stessa risorsa R1. Supponiamo inoltre che i thread A e B hanno una priorità più alta di C, quindi hanno l'attributo priority con valore più alto rispetto a C.

Finchè A e B tentano di acquisire la risorsa R1, ovviamente C non può effettuare il lock e non la può utilizzare, perché A e B hanno la priorità più alta rispetto a C.

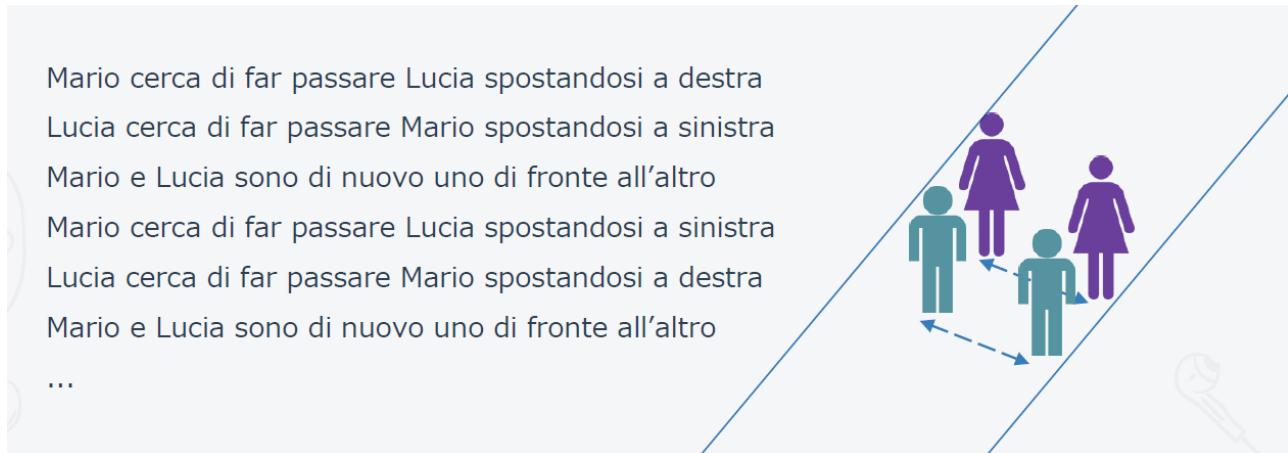
In questo caso C è soggetto a starvation.

LIVELOCK

Il livelock si ha quando 2 thread sono impegnati a rispondersi reciprocamente e non sono in grado di proseguire nell'esecuzione del task.

In questo caso i thread non sono bloccati come accade nel deadlock, ma sono semplicemente occupati a fare altro.

L'esempio classico è quello di Mario e Lucia, che si incontrano sul marciapiede, uno di fronte all'altro.



Ovviamente, se uno tra Mario e Lucia non cambia strategia, non si andrà avanti.

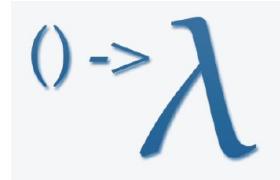
Quando uno dei due si arresta e fa passare l'altro, a quel punto si ha lo sblocco della situazione.

ESPRESSIONI LAMBDA

INTRODUZIONE ALLE ESPRESSIONI LAMBDA

Le espressioni lambda sono disponibili a partire dalla versione 8 di JAVA. L'introduzione di questa funzionalità ha consentito a JAVA di colmare il gap che aveva con altri linguaggi.

In JAVA, un'espressione lambda non è altro che una funzione, in particolare una funzione anonima.



Per funzione anonima si intende una funzione senza dichiarazione, quindi una funzione che non ha nome. Da quando sono state introdotte in JAVA le espressioni lambda, le funzioni anonime sono considerate come un nuovo tipo di dato.

Un'espressione lambda può essere :

- passata come argomento di un metodo;
- essere restituita in uscita da un metodo.

```
myvar.myMethod(e -> System.out.println("Lorem ipsum..."));
```

Le espressioni lambda hanno particolare utilità quando dobbiamo definire una funzione breve (con poche righe di codice), che verrà utilizzata solo una volta.

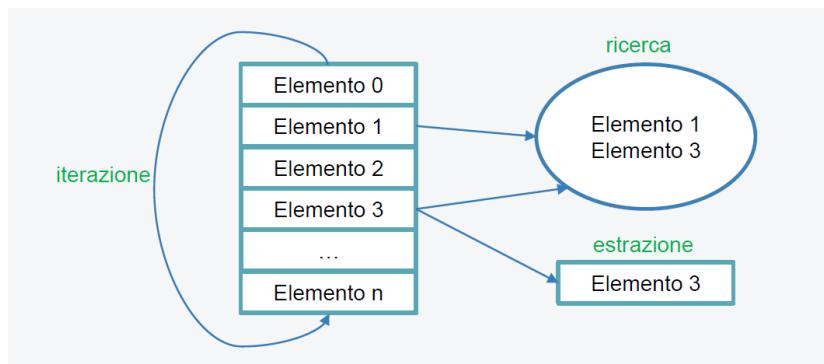
Chiaramente, se dobbiamo definire un metodo che deve essere disponibile in più classi, o ad altri software, dobbiamo creare un metodo normale e non un'espressione lambda.

```
public class EsempiLambdaExpression {  
    public static void main(String[] args) {  
        Thread t1 = new Thread() {  
            @Override  
            public void run() {  
                System.out.println("Ecco un thread creato senza Lambda Expressions");  
            }  
        };  
  
        Thread t2 = new Thread(() -> System.out.println("Questo è un thread creato usando le Lambda Expressions...!"));  
  
        t1.start();  
        t2.start();  
    }  
}
```

In questo esempio possiamo vedere due casi:

- nel primo caso, il thread t1 è il caso classico di utilizzo di un thread, in cui abbiamo l'implementazione del metodo run() all'interno dell'istanza Thread;
- nel secondo caso, invece, al thread t2 abbiamo passato in ingresso direttamente un'espressione lambda.

Le espressioni lambda hanno consentito di migliorare notevolmente la gestione delle liste, perché attraverso tali espressioni viene resa più semplice l'iterazione, la ricerca e l'estrazione di dati da una lista.



Per poter capire bene come utilizzare le espressioni lambda in JAVA, è necessario comprendere cosa sono le interfacce funzionali o Functional Interfaces.

Un'interfaccia funzionale è un'interfaccia che definisce un solo metodo e viene identificata in JAVA mediante l'annotation `@FunctionalInterface`.

Quando creiamo un'interfaccia di tipo funzionale, questa interfaccia può essere implementata da un'espressione lambda, che effettua l'override del metodo.

Vediamo un esempio:

```
@FunctionalInterface  
public interface EsempioFunctionalInterfaceFormaGeometrica {  
    public double calcolaArea(double lato1, double lato2);  
}
```

Questa interfaccia funzionale semplicemente definisce un metodo `calcolaArea()`, che prende in ingresso `lato1` e `lato2`.

A questo punto, posso utilizzare questa interfaccia funzionale per creare un oggetto di tipo `FormaGeometrica`.

Applichiamo tale esempio prima di Java 8:

Creo un'interfaccia `FormaGeometrica` che definisce il metodo `calcolaArea(...)`:

```
public interface FormaGeometrica {  
    public double calcolaArea(double lato1, double lato2);  
}
```

Creo una classe che implementa la nostra interfaccia...

```
public class Rettangolo implements FormaGeometrica {  
    @Override  
    public double calcolaArea(double lato1, double lato2) {  
        return lato1*lato2;  
    }  
}
```

A questo punto, posso utilizzare la nuova classe per creare un oggetto di tipo `FormaGeometrica`

```
FormaGeometrica r1 = new Rettangolo();  
r1.calcolaArea(3, 4);
```

Prima di JAVA 8 avevo una semplice interfaccia FormaGeometrica, con il metodo definito calcolaArea(), creavo una classe Rettangolo che implementava il metodo definito in FormaGeometrica e, quando serviva, potevo creare un'istanza della classe Rettangolo per invocare il metodo calcolaArea().

Applicando questo esempio dopo JAVA 8:

EsempioFunctionalInterfaceFormaGeometrica.java

```
@FunctionalInterface  
public interface EsempioFunctionalInterfaceFormaGeometrica {  
    public double calcolaArea(double lato1, double lato2);  
}
```

Main.java

```
public class Main {  
  
    public static void main(String[] args) {  
        EsempioFunctionalInterfaceFormaGeometrica Rettangolo = (a,b) -> a * b;  
  
        Rettangolo.calcolaArea(3, 4);  
    }  
}
```

abbiamo un'interfaccia funzionale FormaGeometrica che, anche qui, definisce il metodo calcolaArea(), dopodichè nel main() ho creato un oggetto di tipo Rettangolo, assegnandogli un'espressione lambda e, arrivati a questo punto, posso di nuovo invocare il metodo calcolaArea().

Come si può notare dall'ultimo esempio, non ho bisogno di creare una classe che implementi l'interfaccia funzionale, ma utilizzo direttamente l'interfaccia funzionale creando al volo un oggetto di tipo Rettangolo.

Questo tipo di approccio utilizzato da JAVA 8 in poi è molto più snello dal punto di vista del codice.

SINTASSI DELLE ESPRESSIONI LAMBDA

La sintassi delle espressioni lambda è la seguente:

```
(Lista argomenti) -> { istruzioni...; }  
oppure  
(Lista argomenti) -> espressione
```

Esempi

Espressione lambda senza argomenti in ingresso che stampa la frase «Lorem ipsum...»
() -> System.out.println("Lorem ipsum...")

Espressione lambda che prende in ingresso due numeri interi e restituisce il prodotto
(int a, int b) -> a*b

Espressione lambda che prende in input una stringa e restituisce il suo contenuto convertito in maiuscolo
(String nome) -> nome.toUpperCase()

Espressione lambda che prende in input due stringhe ed esegue le istruzioni contenute tra {...}
(String nome, String cognome) -> {
 String frase = "Ciao " + nome + " " + cognome;
 System.out.println(frase);
}

La sintassi che abbiamo appena visto è particolarmente utile quando abbiamo a che fare con funzioni di ordine superiore, ossia funzioni che prendono in ingresso un'altra funzione.

Le funzioni di ordine superiore sono utilizzate per:

- ordinare liste o collezioni;
- eseguire un task particolare che è già stato pianificato;
- e così via... .

Prima delle espressioni lambda, in JAVA era possibile utilizzare delle classi dedicate (Comparator, Runnable, ...).

Con le espressioni lambda non abbiamo più bisogno di utilizzare queste classi dedicate, rendendo il codice più snello.

Continuando il tema di sintassi delle espressioni lambda, parliamo delle istruzioni break e continue.

Il break e il continue:

- non si possono utilizzare all'interno del blocco {...};
- si possono utilizzare all'interno di un ciclo che è presente all'interno del blocco {...}.

Vediamo un esempio:

```
public class EspressioniLambdaConBreakEContinue {  
  
    public static void main(String[] args) {  
  
        Thread t1 = new Thread(()->{  
            System.out.println("Sono nel thread 1");  
  
            int max = 0;  
  
            while(max < 10) {  
                System.out.println("max = " + max);  
  
                if (max == 5) {  
                    break;  
                }  
            }  
        });  
    }  
}
```

Il corpo di un'espressione lambda restituisce un output (ritorna un valore), se nel codice abbiamo delle iterazioni (for, while, ...), oppure degli statement di controllo (if, switch, ...). Nello specifico, all'interno del corpo di un'espressione lambda, ogni ramo del codice deve ritornare un valore o lanciare un'eccezione, altrimenti avremo problemi di compilazione.

Vediamo un esempio:

Esempio

```
private static void saluta(String nome, String cognome) {  
    Callable<String> callMe = () -> {  
        if(nome == null && cognome == null) {  
            throw new Exception("Il nome ed il cognome non sono stati inseriti");  
        } else if(nome == null && cognome != null) {  
            throw new Exception("Il nome non è stato inserito");  
        } else if(nome != null && cognome == null) {  
            throw new Exception("Il cognome non è stato inserito");  
        } else {  
            return "Ciao " + nome + " " + cognome;  
        }  
    };  
    ...  
}
```

Se ometto di inserire un throw o un **return** in un ramo dello statement **if**, la classe non viene compilata!

Provate a togliere questa istruzione sostituendola lasciando vuoto l'else if...

Infine, sempre in tema di sintassi delle espressioni lambda, è possibile:

- omettere il tipo dei parametri in ingresso;
- omettere le parentesi se c'è solo un parametro in ingresso.

PACKAGE JAVA.UTIL.FUNCTION

Il package `java.util.function`, disponibile a partire dalla versione JAVA 8, contiene un set di interfacce funzionali standard, tra cui:

- `Predicate`;
- `Consumer`;
- `Supplier`;
- `Function`;
- `UnaryOperator`;
- `BinaryOperator`;
-

JAVA.UTIL.FUNCTION.PREDICATE

L'interfaccia funzionale Predicate rappresenta un predicato di un elemento.

Per capire a cosa serve questa interfaccia, ricordiamo cosa è un predicato. Un predicato è una proposizione che contiene delle variabili, il cui valore determina la veridicità della proposizione.

Esempi:

- «**a** è multiplo di 3»
- «**s** è una stringa vuota»
- «**a** è maggiore di **b**»

Il metodo definito all'interno dell'interfaccia Predicate è il metodo test(T t), che prende in ingresso un tipo generico e restituisce un risultato booleano.

Vediamo un esempio:

Utente.java

```
package espressioni_lambda;

public class Utente {
    private String nome;
    private String cognome;
    private int eta;
    private String cittaResidenza;
    private String email;
    private String password;

    public Utente(String nome, String cognome, int eta, String cittaResidenza, String email, String password) {
        this.nome = nome;
        this.cognome = cognome;
        this.eta = eta;
        this.cittaResidenza = cittaResidenza;
        this.email = email;
        this.password = password;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getCognome() {
        return cognome;
    }

    public void setCognome(String cognome) {
```

```

        this.cognome = cognome;
    }

    public int getEta() {
        return eta;
    }

    public void setEta(int eta) {
        this.eta = eta;
    }

    public String getCittaResidenza() {
        return cittaResidenza;
    }

    public void setCittaResidenza(String cittaResidenza) {
        this.cittaResidenza = cittaResidenza;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
}

```

MainUtente.java

```

import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;

public class MainUtente {

    private List<Utente> elencoUtenti() {
        List<Utente> utenti = new ArrayList<Utente>();

        utenti.add(new Utente("Paolo", "Preite", 39, "Roma", "info@paolopreite.it",
"test"));
        utenti.add(new Utente("Mario", "Rossi", 40, "Roma", "info@paolopreite.it",
"test"));
        utenti.add(new Utente("Antonio", "Di Girolamo", 23, "Roma",
"info@paolopreite.it", "test"));
        utenti.add(new Utente("Caterina", "Montefalco", 55, "Roma",
"info@paolopreite.it", "test"));
        utenti.add(new Utente("Valeria", "Natelli", 45, "Roma", "info@paolopreite.it",
"test"));
        utenti.add(new Utente("Giovanna", "D'Antonelli", 50, "Roma",
"info@paolopreite.it", "test"));
    }
}

```

```

        "info@paolopreite.it", "test")));
        utenti.add(new Utente("Paolo", "Pisani", 21, "Roma", "info@paolopreite.it",
"test"));
        utenti.add(new Utente("Laura", "Gambaro", 19, "Roma", "info@paolopreite.it",
"test"));
        utenti.add(new Utente("Benedetto", "Satini", 38, "Roma", "info@paolopreite.it",
"test"));

        return utenti;
    }

    public List<Utente> cercaUtenti(List<Utente> utenti, Predicate<Utente> p){
        List<Utente> utentiTrovati = new ArrayList<Utente>();

        for (Utente u:utenti) {
            if (p.test(u)) {
                utentiTrovati.add(u);
            }
        }
        return utentiTrovati;
    }
}

```

La classe Utente rappresenta un generico utente, ed ha il suo costruttore. All'interno della nostra classe Main, invece, abbiamo un metodo cercaUtenti(), che prende in ingresso una lista di utenti e un predicato. All'interno del metodo cercaUtenti() abbiamo una nuova lista utentiTrovati, che conterrà tutti gli utenti che soddisfano il criterio di ricerca definito nel predicato in ingresso. Infine abbiamo un metodo elencoUtenti(), che contiene tutti gli utenti che dovranno essere sottoposti al criterio di ricerca.

Come si può notare però dall'esempio non è stato ancora espresso un criterio di ricerca, ma saremo noi a passarlo in ingresso nel seguente modo:

MainUtente.java

```

import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;

public class MainUtente {

    .....
    .....

    public static void main(String[] args) {
        MainUtente mainUtente = new MainUtente();
        List<Utente> elenco = mainUtente.elencoUtenti();

        List<Utente> trovati = mainUtente.cercaUtenti(elenco, utente ->
utente.getNome().equals("Paolo"));

        for (Utente utenteTrovato : trovati) {
            System.out.println(utenteTrovato.getCognome() + " " +
utenteTrovato.getNome()); /*Preite Paolo, Pisani Paolo*/
        }
    }
}

```

JAVA.UTIL.FUNCTION.CONSUMER

L'interfaccia funzionale Consumer rappresenta un'operazione che accetta un argomento in ingresso e non ritorna alcun valore. Viene utilizzato generalmente per svolgere dei piccoli task.

Il metodo messo a disposizione da questa interfaccia è il metodo accept(T t), che prende in ingresso un tipo generico e non restituisce niente, quindi è un metodo void.

Vediamo un esempio che prende in considerazione le due classi utilizzate nell'esempio del paragrafo precedente, ossia Utente e MainUtente:

MainUtente.java

```
import java.util.ArrayList;
import java.util.List;
import java.util.function.Consumer;
import java.util.function.Predicate;

public class MainUtente {
.....
.....
public static void main(String[] args) {
.....
.....
    Consumer<Utente> consumer = utente -> System.out.println(utente.getCognome() +
" " + utente.getNome());
    elenco.forEach(consumer); /*Preite Paolo
                                Rossi Mario
                                Di Girolamo Antonio
                                Montefalco Caterina
                                Natelli Valeria
                                D'Antonelli Giovanna
                                Pisani Paolo
                                Gambaro Laura
                                Satini Benedetto*/
}
}
```

In questo esempio abbiamo visto come iterare una lista con l'interfaccia Consumer.

JAVA.UTIL.FUNCTION.SUPPLIER

L'interfaccia funzionale Supplier rappresenta l'esatto contrario del Consumer, ossia non accetta argomenti in ingresso e ritorna un valore.

Il metodo messo a disposizione da questa interfaccia è il metodo T get(), che non prende niente in ingresso e restituisce un valore.

Vediamo un esempio che prende in considerazione le due classi utilizzate nell'esempio del paragrafo JAVA.UTIL.FUNCTION.PREDICATE, ossia Utente e MainUtente:

MainUtente.java

Esempio – Stampa degli elementi di una lista

```
List<Utente> utenti = m.elencoUtenti();

for (Utente utente : utenti) {
    m.stampaDatiUtente(() -> utente);
}

private void stampaDatiUtente(Supplier<Utente> u) {
    System.out.println(
        u.get().getCognome() + " " + u.get().getNome() + " " +
        u.get().getEmail());
}
```

1. Recupero una lista di utenti
2. Itero la lista ed invoco il metodo *stampaDatiUtente(...)* che ho creato e che prende in ingresso un Supplier
3. Nel metodo *stampaDatiUtente(...)*, utilizzo il metodo *get()* del Supplier

Rimaniamo sempre nel caso di stampa dell'elenco degli utenti. Abbiamo definito un metodo *stampaDatiUtente()*, che prende in ingresso un Supplier di tipo Utente. A questo punto invochiamo, all'interno del metodo *stampaDatiUtente()*, il metodo *get()* messo a disposizione dall'interfaccia Supplier. Tale metodo *get()* ci consentirà di accedere a tutti gli elementi della classe Utente. La lista utente va iterata in un for e, ad ogni iterazione, va invocato il metodo *stampaDatiUtente()*, che prenderà semplicemente in input l'espressione lambda senza parametri in ingresso.

JAVA.UTIL.FUNCTION.FUNCTION

L'interfaccia funzionale Function rappresenta una funzione che accetta in ingresso un argomento e produce un risultato, ritornando un valore.

Il metodo messo a disposizione da questa interfaccia è il metodo R apply(T t), che prende in ingresso un parametro e restituisce un risultato R.

L'interfaccia Function è utilizzata quando dobbiamo passare in input ad un metodo un blocco di codice.

Vediamo un esempio:

Esempio – Calcolo di un'operazione

```
private void calcolaOperazione(double operando, Function<Double, Double> funzione){  
    double risultato = funzione.apply(operando);  
    System.out.println("Il risultato dell'operazione è " + risultato);  
}  
  
public static void main(String[] args) {  
    Main m = new Main();  
  
    double a = 12;  
    m.calcolaOperazione(a, val -> val * val);  
    m.calcolaOperazione(a, val -> Math.sqrt(val));  
}
```

1. Definisco un metodo che prende in ingresso un numero ed una funzione e stampa il risultato dell'operazione eseguita
2. Utilizzo questa funzione per stampare il quadrato e la radice quadrata del numero passato in ingresso

Nell'esempio, tramite il metodo apply() viene applicata la funzione, passata in ingresso al metodo calcolaOperazione(), sull'operando specificato e viene ritornato il risultato.

JAVA.UTIL.FUNCTION.UNARYOPERATOR

L'interfaccia UnaryOperator rappresenta un'operazione su un operando e produce un risultato dello stesso tipo dell'operando.

Il metodo T apply(T t), che è il metodo definito nell'interfaccia UnaryOperator, prende in ingresso un elemento di tipo generico T e ritorna un elemento che ha lo stesso tipo T.

Esempi

```
UnaryOperator<String> unop = str -> str.toLowerCase();  
System.out.println(unop.apply("Prova Di Stampa Minuscolo"));
```

```
UnaryOperator<Long> unop2 = val -> val*val;  
long num = 10;  
System.out.println("Il quadrato di " + num + " è " + unop2.apply(num));
```

JAVA.UTIL.FUNCTION.BINARYOPERATOR

L'interfaccia BinaryOperator rappresenta un'operazione su due operandi e produce un risultato dello stesso tipo degli operandi.

Anche qui abbiamo il metodo T apply(T t1, T t2), che prende in ingresso due operandi di tipo T e restituisce un risultato dello stesso tipo T.

Esempi

```
BinaryOperator<Double> biop = (a1, a2) -> a1 * a2;

double x = 10.5;
double y = 15;

System.out.println("La moltiplicazione tra " + x + " ed " + y + " è: " + biop.apply(x, y));

BinaryOperator<String> biop2 = (s1, s2) -> "Ciao " + s1 + " " + s2;

utenti = m.elencoUtenti();

for (Utente utente : utenti) {
    System.out.println(biop2.apply(utente.getNome(), utente.getCognome()));
}
```

ESEMPIO CASO D'USO CON LE LAMBDA: FILTRARE UNA LISTA IN BASE A DEI CRITERI

```
import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;

public class EsempioCasoDUssoFiltrareRicercaUtentiInUnaListaInBaseACriteriConLeLambda {

    public static void main(String[] args) {
        EsempioCasoDUssoFiltrareRicercaUtentiInUnaListaInBaseACriteriConLeLambda
        esempioCercaUtenti = new
        EsempioCasoDUssoFiltrareRicercaUtentiInUnaListaInBaseACriteriConLeLambda();

        /* Ricerca utenti con cognome Preite */
        List<Utente> trovati =
        esempioCercaUtenti.cercaUtenti(esempioCercaUtenti.elencoUtenti(), utente ->
        utente.getCognome().equals("Preite"));

        System.out.println("utenti con cognome Preite");
        System.out.println();

        for (Utente utente : trovati) {
            System.out.println(utente.getCognome() + " " + utente.getNome());
        }

        System.out.println(" ----- ");

        /* Ricerca utenti con email .it */
    }
}
```

```

        trovati = esempioCercaUtenti.cercaUtenti(esempioCercaUtenti.elencoUtenti(),
utente -> utente.getEmail().contains(".it"));

        System.out.println("utenti email .it");
        System.out.println();

        for (Utente utente : trovati) {
            System.out.println(utente.getCognome() + " " + utente.getNome());
        }

        System.out.println(" ----- ");

        /* Ricerco utenti con età > 40 */
        trovati = esempioCercaUtenti.cercaUtenti(esempioCercaUtenti.elencoUtenti(),
utente -> utente.getEta() > 40);

        System.out.println("utenti con eta' > 40");
        System.out.println();

        for (Utente utente : trovati) {
            System.out.println(utente.getCognome() + " " + utente.getNome());
        }

        System.out.println(" ----- ");

        /* Ricerco un utente in particolare */
        Utente u =
esempioCercaUtenti.cercaUtenteSullaBaseDiAlcuniCriteri(esempioCercaUtenti.elencoUtenti(
), utente -> utente.getCittaResidenza().equals("Roma"));

        System.out.println("Ricerca utente singolo:");
        System.out.println();

        if (u != null){
            System.out.println(u.getCognome() + " " + u.getNome());
        }

        System.out.println(" ----- ");
    }

    public List<Utente> cercaUtenti(List<Utente> elencoUtenti, Predicate<Utente>
criterio) {
        List<Utente> trovati = new ArrayList<Utente>();

        for (Utente utente : elencoUtenti) {
            if (criterio.test(utente)) {
                trovati.add(utente);
            }
        }
        return trovati;
    }

    public Utente cercaUtenteSullaBaseDiAlcuniCriteri(List<Utente> elencoUtenti,
Predicate<Utente> criterio) {

        for (Utente utente : elencoUtenti){

```

```

        if (criterio.test(utente)) {
            return utente;
        }
    }
    return null;
}

private List<Utente> elencoUtenti() {
    List<Utente> utenti = new ArrayList<Utente>();

    utenti.add(new Utente("Paolo", "Preite", 39, "Lecce", "info@paolopreite.it",
"test"));
    utenti.add(new Utente("Mario", "Rossi", 40, "Roma", "info@email.it", "test"));
    utenti.add(new Utente("Antonio", "Di Girolamo", 23, "Milano", "info@prova.it",
"test"));
    utenti.add(new Utente("Caterina", "Montefalco", 55, "Parma", "info@test.it",
"test"));
    utenti.add(new Utente("Valeria", "Natelli", 45, "Pavia", "info@natelli.it",
"test"));
    utenti.add(new Utente("Giovanna", "D'Antonelli", 50, "Modena", "info@anto.it",
"test"));
    utenti.add(new Utente("Paolo", "Pisani", 21, "Napoli", "info@prova2.it",
"test"));
    utenti.add(new Utente("Laura", "Gambaro", 19, "Enna", "info@gambaro.it",
"test"));
    utenti.add(new Utente("Benedetto", "Satini", 38, "Palermo", "info@hosting.it",
"test"));
    return utenti;
}
}

```

Output del programma:

utenti con cognome Preite

Preite Paolo

utenti email .it

Preite Paolo

Rossi Mario

Di Girolamo Antonio

Montefalco Caterina

Natelli Valeria

D'Antonelli Giovanna

Pisani Paolo

Gambaro Laura

Satini Benedetto

utenti con eta' > 40

Montefalco Caterina

Natelli Valeria

D'Antonelli Giovanna

Ricerca utente singolo:

Rossi Mario

ESPRESSIONI REGOLARI

L'espressione regolare è uno strumento che ci consente di lavorare in maniera avanzata sulle stringhe.

Essa si può definire come una stringa più o meno complessa che, attraverso una sintassi particolare, consente di definire delle regole di individuazione degli elementi all'interno di un testo.

In particolare, un'espressione regolare consente di:

- convalidare i dati;
- ricerche all'interno di un testo;
- ...

A partire dalla versione 1.4 di JAVA, tutte le classi che gestiscono le espressioni regolari si trovano nel package `java.util.regex`.

Le espressioni regolari non sono una caratteristica tipica di JAVA, ma sono una caratteristica generale che viene utilizzata in tutti i linguaggi di programmazione.

Per JAVA, le classi che consentono di lavorare con le espressioni regolari sono:

- Pattern;
- PatternSyntaxException;
- Matcher.

SINTASSI DELLE ESPRESSIONI REGOLARI

Le regole di base per definire un'espressione regolare sono:

- [...] cerca in una stringa tutti i caratteri che sono inseriti tra [];
- [^...] cerca in una stringa tutti i caratteri che non sono inseriti tra [];
- [...]...] cerca in una stringa tutti i caratteri compresi nell'intervallo ...-... ;
- [...&&...] cerca in una stringa solo i caratteri che sono presenti nel primo e nel secondo insieme contemporaneamente (intersezione di insiemi);
- . cerca in una stringa qualsiasi carattere;
- [...]...] cerca in una stringa tutti i caratteri del primo e del secondo insieme (unione di insiemi);
- ^[...] cerca i caratteri indicati in [] che si trovano solo all'inizio della linea;
- [...]\$ cerca i caratteri indicati in [] che si trovano solo alla fine della linea;
- a* cerca 0 o più occorrenze di <<a>> dell'espressione regolare <<a>>;
- a{n} cerca il numero esatto <<n>> di occorrenze dell'espressione regolare <<a>>;
- a{n,} cerca almeno <<n>> occorrenze dell'espressione regolare <<a>>;
- a{n,m} cerca almeno <<n>>, ma non più di <<m>>, occorrenze dell'espressione regolare <<a>>.

Oltre a queste regole, abbiamo anche delle forme abbreviate per rappresentare un insieme di caratteri:

- \d corrisponde a [0-9], quindi tutti i numeri che vanno da 0 a 9;
 - \D corrisponde a [^0-9], quindi tutti i caratteri che non sono numeri;
 - \s identifica il carattere spazio;
 - \S corrisponde a [^\s], quindi tutti i caratteri tranne lo spazio;
 - \w corrisponde a [a-zA-Z_0-9], quindi tutti i caratteri (maiuscoli e minuscoli) alfanumerici;
 - \W corrisponde a [^\w], quindi tutti i caratteri speciali, compreso lo spazio. Sono esclusi i caratteri alfanumerici.

Vediamo un esempio:


```

System.out.println(testo.replaceAll("[a-z]", "K"));

/* Output:
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore
et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut
ex ea commodo consequat. Dius aute irure dolor in reprehenderit in voluptate velit esse cillum dolore
mollit anim id est laborum.
LLLLLLLLLLLLLLLLLoLoLore
1974er1222
$%AE/())=K
*/


System.out.println("-----");
System.out.println("a*");
System.out.println("Sostituisce nel testo tutte le occorrenze 'Lo' con 'H'");
System.out.println(testo.replaceAll("Lo*", "H"));

/* Output:
Hrem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore
et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut
ex ea commodo consequat. Dius aute irure dolor in reprehenderit in voluptate velit esse cillum dolore
mollit anim id est laborum.
HHHHHHHHHHHHHHHHHHHHHHHHrem
1974er1222
$%AE/())=m
*/
System.out.println("-----");
System.out.println("a{n}");
System.out.println("Sostituisce nel testo 'LL' con 'H'");
System.out.println(testo.replaceAll("L{2}", "H"));

/* Output:
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore
et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut
ex ea commodo consequat. Dius aute irure dolor in reprehenderit in voluptate velit esse cillum dolore
mollit anim id est laborum.
HHHHHHHHHLLoLoLore
1974er1222
$%AE/())=m
*/
System.out.println("-----");
System.out.println("a{n,}");
System.out.println("Sostituisce nel testo le occorrenze che presentano un numero di L attaccate almeno = 2 con 'H'");
System.out.println(testo.replaceAll("L{2,}", "H"));

/* Output:
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore
et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut
ex ea commodo consequat. Dius aute irure dolor in reprehenderit in voluptate velit esse cillum dolore
mollit anim id est laborum.
HoLoLore
1974er1222
$%AE/())=m
*/
System.out.println("-----");
System.out.println("a{n,m}");
System.out.println("Sostituisce nel testo le occorrenze che presentano un numero di L attaccate almeno = 2, ma non
maggiore di 4, con 'H'");
System.out.println(testo.replaceAll("L{2,4}", "H"));

/* Output:
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore
et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut
ex ea commodo consequat. Dius aute irure dolor in reprehenderit in voluptate velit esse cillum dolore
mollit anim id est laborum.
HHHHHHoLoLore
1974er1222
$%AE/())=m
*/

```


CLASSE PATTERN

La classe Pattern rappresenta in JAVA la versione compilata di un'espressione regolare. Un'espressione regolare, abbiamo visto già, che è una stringa (sequenza di caratteri) che ha una sua logica. Questa sequenza di caratteri, attraverso la classe Pattern, viene compilata in versione bytecode.

I metodi principali della classe Pattern sono:

- `compile(String regex)`, che prende in ingresso l'espressione regolare, la compila e restituisce un oggetto di tipo Pattern che contiene tale espressione regolare;
 - `compile(String regex, int flags)`, che è la variante del metodo precedente, con l'aggiunta della variabile flags. La variabile flags è un'opzione che consente di modificare un comportamento del match;
 - `flags()`, che restituisce il flag impostato in fase di creazione dell'oggetto Pattern;
 - `matcher(CharSequence input)`, che prende in ingresso un testo input, lo elabora e, sulla base dell'espressione regolare, restituisce un oggetto di tipo Matcher che possiamo utilizzare per manipolare la stringa;
 - `matches(String regex, CharSequence input)`, che analizza la stringa input e restituisce true se il match con l'espressione regolare regex ha successo, false altrimenti;
 - `pattern()`, che restituisce l'espressione regolare inserita in fase di creazione dell'oggetto;
 - `split(CharSequence input)`, che suddivide la stringa input in più stringhe, utilizzando come separatore l'espressione regolare indicata in fase di creazione dell'oggetto Pattern.

Vediamo un esempio:

```
import java.util.regex.Pattern;

public class EsempioPattern {

    public static void main(String[] args) {
        /* Creazione dell'istanza Pattern */
        Pattern p = Pattern.compile("\d");

        /*-----*/
        String elenco = "1. Juventus, 2. Roma, 3. Napoli, 4. Atalanta, 5. Lazio";

        /* split() suddivide la stringa su cui viene richiamato,
        basandosi sull'espressione regolare specificata nel compile(),
        e restituisce un array di stringhe */
        String[] elencoArray = p.split(elenco);

        for (int i = 1; i < elencoArray.length; i++) {
            System.out.println(elencoArray[i].replaceAll(", ", ""));
            /* . Juventus
             * . Roma
             * . Napoli
             * . Atalanta
             * . Lazio */
        }
        System.out.println();
        /*-----*/

        /*-----*/
        String testo = "info@paolopreite.it";

        /* matches() ritorna true se l'espressione regolare in input si trova
        * all'interno del testo, altrimenti false */
        boolean match = Pattern.matches(".@.*", testo);
        System.out.println("il testo contiene la @? " + match); // true
        System.out.println();
        /*-----*/

        /*-----*/
        /* pattern() restituisce l'espressione regolare
        specificata nel compile()*/
        System.out.println(p.pattern()); // \d
        System.out.println();
        /*-----*/

        /*-----*/
        /* Il flag CASE_INSENSITIVE permette di trascurare
        * se una lettera è maiuscola o minuscola*/
        p = Pattern.compile("Paolo", Pattern.CASE_INSENSITIVE);

        String[] elencoArray2 = p.split(testo);

        for (int i = 0; i < elencoArray2.length; i++) {
            System.out.println(elencoArray2[i]); /* info@
                                                 preite.it */
        }
        System.out.println();
        /*-----*/
```

```

/*-----*/
/*flags() restituisce il valore intero del flag utilizzato nel compile.
 * In questo caso il flag CASE_INSENSITIVE vale 2*/
System.out.println(p.flags()); // 2;
/*-----*/
}

}

```

CLASSE MATCHER

La classe Matcher è una classe che consente di effettuare il match tra una espressione regolare e una stringa.

Per creare un'istanza della classe Matcher si deve utilizzare il metodo matcher(CharSequence input), disponibile nella classe Pattern, che prende in ingresso la stringa su cui applicare l'espressione regolare definita nell'oggetto Pattern

Vediamo nello specifico la sintassi per creare un oggetto Matcher:

```
Pattern p = Pattern.compile("espressione regolare");
Matcher m = p.matcher("stringa da elaborare");
```

Vediamo un esempio:

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class EsempioMatcher {

    public static void main(String[] args) {
        String text =
            "<h1>Titolo 1</h1>"+
            "<p>Testo A</p>"+
            "</hr>"+
            "<p>Testo B</p>"+
            "<h2>Titolo 2</h2>"+
            "<p>Testo C</p>";

        String regex = "(<p>([<^>]+)</p>)";

        Pattern pattern = Pattern.compile(regex);

        /* L'oggetto Matcher applica l'espressione
         * regolare regex sulla stringa text */
        Matcher matcher = pattern.matcher(text);
    }
}
```

Prima di vedere quali sono i metodi utili nella classe Matcher, vediamo cosa sono i gruppi.

I gruppi sono espressioni regolari racchiuse tra parentesi tonde, come ad esempio A(B(C))D.

L'esempio A(B(C))D si può definire nei seguenti gruppi:

- Gruppo 0 = ABCD (tutta l'espressione regolare);
- Gruppo 1 = BC (l'espressione regolare che si trova all'interno delle prime parentesi tonde);
- Gruppo 2 = C.

Vediamo adesso quali sono i metodi disponibili nella classe Matcher:

- find(), che cerca la sottosequenza di caratteri successiva individuata nel testo;
- find(int start), che è analogo al metodo precedente, con la differenza che la ricerca viene effettuata partendo dall'indice specificato in ingresso;
- group(), che ritorna la sottosequenza individuata nel testo che combacia con l'intera espressione regolare, cioè con il gruppo 0;
- group(int group), che ritorna la sottosequenza individuata nel testo che combacia con l'espressione regolare contenuta nel gruppo specificato. Ad esempio, se l'espressione regolare è A(B(C))D, group(1) ritorna la sottosequenza individuata nel testo che combacia con l'espressione regolare BC;
- groupCount(), che restituisce il numero di gruppi presenti nell'espressione regolare. Da notare che nel conteggio dei gruppi dell'espressione regolare, il gruppo 0 non viene considerato;
- pattern(), che restituisce l'oggetto Pattern da cui è stato creato il matcher;
- replaceAll(String replacement), che sostituisce tutte le sottosequenze individuate con la stringa passata in ingresso;
- replaceFirst(String replacement), che sostituisce solo la prima sottosequenza individuata con la stringa passata in ingresso;
- reset(), che effettua il reset del matcher;
- reset(CharSequence input), che effettua il reset, sostituendo la stringa attuale con una nuova stringa passata in ingresso.

Vediamo un esempio:

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class EsempioMatcher {

    public static void main(String[] args) {
        String text =
            "<h1>Titolo 1</h1>" +
            "<p>Testo A</p>" +
            "</hr>" +
            "<p>Testo B</p>" +
            "<h2>Titolo 2</h2>" +
            "<p>Testo C</p>";
```

```

/*L'espressione regolare (<p>([<^>]+)</p>) è composta dai seguenti gruppi:
- Gruppo 0 : (<p>([<^>]+)</p>);
- Gruppo 1 : <p>([<^>]+)</p>;
- Gruppo 2 : [<^>]+ */
String regex = "<p>([<^>]+)</p>";

Pattern pattern = Pattern.compile(regex);

/* L'oggetto Matcher applica l'espressione
regolare regex sulla stringa text */
Matcher matcher = pattern.matcher(text);

System.out.println("Gruppi presenti nell'espressione regolare " +
    regex + ": " + matcher.groupCount()); // Gruppi presenti
nella espressione regolare (<p>([<^>]+)</p>): 2

System.out.println();

/*Finche' trovo occorrenze dell'espressione regolare regex in text...*/
while (matcher.find()) {

    /* stampa tutto ciò che nel text corrisponde all'intera espressione
    regolare regex (gruppo 0)*/
    System.out.println(matcher.group()); /* <p>Testo A</p>
                                         <p>Testo B</p>
                                         <p>Testo C</p> */

    /* stampa tutto ciò che nel text corrisponde al gruppo 1 dell'espressione
    regolare regex*/
    System.out.println(matcher.group(1)); /* <p>Testo A</p>
                                         <p>Testo B</p>
                                         <p>Testo C</p> */

    /* stampa tutto ciò che nel text corrisponde al gruppo 2 dell'espressione
    regolare regex*/
    System.out.println(matcher.group(2)); /* Testo A
                                         Testo B
                                         Testo C */

    System.out.println("-----");
}
}
}

```

APPLICAZIONI PRATICHE DELLE ESPRESSIONI REGOLARI

VALIDAZIONE EMAIL

L'espressione regolare per la validazione delle email è la seguente:

```
String regex = "[a-zA-Z0-9._-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,4}"
```

La parte dell'espressione regolare in cui abbiamo [a-zA-Z0-9._-] indica tutte le lettere minuscole, le lettere maiuscole e i caratteri speciali ._- .

Dopo il +, che concatena la prima parte dell'espressione regolare con il resto, abbiamo la chiocciola @ seguita da un'altra parte di espressione regolare [a-zA-Z0-9.-] . Tale espressione regolare è simile alla prima, solo che esclude il carattere _- .

Dopo il secondo +, abbiamo la terza ed ultima parte di espressione regolare .[a-zA-Z]{2,4}, che considera da un minimo di 2 ad un massimo di 4 lettere maiuscole e minuscole, con il carattere punto compreso.

Di seguito il codice completo:

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class ValidazioneEmail {

    public static void main(String[] args) {

        ValidazioneEmail validazioneEmail = new ValidazioneEmail();
        System.out.println(validazioneEmail.emailFormatValidator("info@paolopreite.it")); // true
        System.out.println(validazioneEmail.emailFormatValidator("paolo.preite@paolopreite.it")); // true
        System.out.println(validazioneEmail.emailFormatValidator("paolo_preite@paolopreite.it")); // true
        System.out.println(validazioneEmail.emailFormatValidator("paolo-preite@paolopreite.it")); // true
        System.out.println(validazioneEmail.emailFormatValidator("paolo-preite@paolopreite.com")); // true
        System.out.println(validazioneEmail.emailFormatValidator("paolo-preite@paolopreite.co.uk")); // true
        System.out.println(validazioneEmail.emailFormatValidator("paolo-preite@paolopreite.name")); // true
        System.out.println(validazioneEmail.emailFormatValidator("paolo%preite@paolopreite.it")); // false
        System.out.println(validazioneEmail.emailFormatValidator("Prova")); // false
        System.out.println(validazioneEmail.emailFormatValidator("info@test.comoi")); // false
        System.out.println(validazioneEmail.emailFormatValidator("@paolopreite.it")); // false
    }

    public boolean emailFormatValidator(String date) {

        String regex = "[a-zA-Z0-9._-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,4}";

        /* Si crea un oggetto Pattern, passandogli l'espressione regolare */
        Pattern pattern = Pattern.compile(regex);

        /* A questo punto, invece, si passa in ingresso l'email*/
        Matcher matcher = pattern.matcher(date);

        /* Il metodo matches ritorna true se la stringa rispetta i criteri dell'espressione regolare, altrimenti false */
        if(matcher.matches()) {
            return true;
        }
        return false;
    }
}
```

VALIDAZIONE FORMATO DATA

L'espressione regolare per la validazione della data è la seguente:

```
String regex = "(0[1-9]|1[2][0-9]|3[01])[-/.](0[1-9]|1[012])[-/.](19|20)\\d\\d";
```

In questa espressione regolare i formati di data ammessi sono:

- dd/mm/yyyy
- dd-mm-yyyy

Dividiamo in parti l'espressione regolare regex per capire come è composta:

- (0[1-9]|1[2][0-9]|3[01]) : considera tutti i numeri che vanno da 01 a 31;
- [- /.] : considera i caratteri / - ;
- (0[1-9]|1[012]) : considera tutti i numeri da 01 a 12;
- (19|20)\\d\\d : considera i numeri dal 1900 al 2099.

Di seguito il codice completo:

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class ValidazioneData {

    public static void main(String[] args) {
        ValidazioneData validazioneData = new ValidazioneData();
        System.out.println(validazioneData.dateFormatValidator("10/12/2016")); // true
        System.out.println(validazioneData.dateFormatValidator("10-12-2016")); // true
        System.out.println(validazioneData.dateFormatValidator("31-12-2016")); // true
        System.out.println(validazioneData.dateFormatValidator("31-12-1899")); // false
        System.out.println(validazioneData.dateFormatValidator("31\\12\\1899")); // false
        System.out.println(validazioneData.dateFormatValidator("2016/02/14")); // false
        System.out.println(validazioneData.dateFormatValidator("12/2016")); // false
        System.out.println(validazioneData.dateFormatValidator("2016/02/14")); // false
        System.out.println(validazioneData.dateFormatValidator("dd/mm/yyyy")); // false
    }

    public boolean dateFormatValidator(String date) {
        String regex = "(0[1-9]|1[2][0-9]|3[01])[-/.](0[1-9]|1[012])[-/.](19|20)\\d\\d";
        /* Si crea un oggetto Pattern, passandogli l'espressione regolare */
        Pattern pattern = Pattern.compile(regex);

        /* A questo punto, invece, si passa in ingresso la data*/
        Matcher matcher = pattern.matcher(date);

        /* Il metodo matches ritorna true se la stringa rispetta i criteri dell'espressione regolare, altrimenti false */
        if(matcher.matches()) {
            return true;
        }
        return false;
    }
}
```

VALIDAZIONE CODICE FISCALE

L'espressione regolare per la validazione del codice fiscale è la seguente:

```
String regex = "[a-zA-Z]{6}\\d\\d[a-zA-Z]\\d\\d[a-zA-Z]\\d\\d\\d[a-zA-Z]";
```

Dividiamo in parti l'espressione regolare regex per capire come è composta:

- [a-zA-Z]{6} : considera un numero massimo di 6 lettere, maiuscole o minuscole che siano;
- \d\d : considera 2 numeri;
- [a-zA-Z] : considera una sola lettera, maiuscola o minuscola che sia.
- \d\d\d : considera 3 numeri

Di seguito il codice completo:

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class ValidazioneCodiceFiscale {

    public static void main(String[] args) {
        ValidazioneCodiceFiscale validazioneCodiceFiscale = new ValidazioneCodiceFiscale();
        System.out.println(validazioneCodiceFiscale.CFFormatValidator("PRTPPP77P15H501K")); // true
        System.out.println(validazioneCodiceFiscale.CFFormatValidator("PRTPPP77P15H501KP001")); // false
        System.out.println(validazioneCodiceFiscale.CFFormatValidator("PRTPPP77P15H")); // false
        System.out.println(validazioneCodiceFiscale.CFFormatValidator("123PPP77P15H501K")); // false
        System.out.println(validazioneCodiceFiscale.CFFormatValidator("PRT12377P15H501K")); // false
        System.out.println(validazioneCodiceFiscale.CFFormatValidator("PRT123AAP15H501K")); // false
    }

    public boolean CFFormatValidator(String date) {
        String regex = "[a-zA-Z]{6}\\d\\d[a-zA-Z]\\d\\d[a-zA-Z]\\d\\d[a-zA-Z]";
        /* Si crea un oggetto Pattern, passandogli l'espressione regolare */
        Pattern pattern = Pattern.compile(regex);

        /* A questo punto, invece, si passa in ingresso il codice fiscale*/
        Matcher matcher = pattern.matcher(date);

        /* Il metodo matches ritorna true se la stringa rispetta i criteri dell'espressione regolare, altrimenti false */
        if(matcher.matches()) {
            return true;
        }
        return false;
    }
}
```

LE DATE

INTRODUZIONE ALLA GESTIONE DELLE DATE IN JAVA

Quando scriviamo un software, capita il più delle volte di avere a che fare con le date e gli orari.

Alcuni esempi di utilizzo delle date sono:

- data di inserimento o modifica di una riga all'interno di una tabella;
- data di creazione di un file;
- data di nascita di un utente;
- data di una fattura;
- ...

Quando lavoriamo con le date, dobbiamo tenere in considerazione diversi fattori, tra cui il formato della data (dd/mm/yyyy o yyyy/mm/dd...) e il fuso orario.

Le classi messe a disposizione da JAVA per la gestione delle date sono:

- `java.sql.Timestamp`;
- `java.util.Date`;
- `java.util.Calendar`;
- `java.util.GregorianCalendar`;
- `java.text.SimpleDateFormat`.

Nella versione 1.1 di JAVA, l'unica classe per la gestione delle date era `java.util.Date`. Molti metodi di questa classe sono stati deprecati già nella versione 1.2, poiché è stata introdotta la classe `java.util.Calendar`.

In JAVA 8 è disponibile una nuova serie di API, che si trovano all'interno del package `java.time`, che ottimizza la gestione delle date e consente di risolvere molte limitazioni presenti nelle vecchie classi.

JAVA.SQL.TIMESTAMP

La classe `Timestamp` è utilizzata per la gestione del tipo di dato `Timestamp`, presente nel database. Il `Timestamp` è un numero, più o meno lungo, che rappresentano anno, mese, giorno, ora, minuti, secondi, millisecondi e nanosecondi di una data.

JAVAUTILDATE

La classe `Date`, come già detto, è stata la prima classe creata per la gestione delle date.

Il suo utilizzo ormai è deprecato a favore della classe `Calendar`. Tuttavia, per motivi di compatibilità, la classe `Date` è ancora presente nella JDK, ma molti dei suoi metodi sono stati comunque deprecati.

Un'istanza della classe `Date` rappresenta un istante di tempo con una precisione al millisecondo.

Vediamo un esempio:

```
import java.util.Date;

public class EsempioDate {

    public static void main(String[] args) {
        EsempioDate esempioDate = new EsempioDate();

        esempioDate.esDate();
    }

    private void esDate() {
        Date data = new Date();

        /* Valore della data in Timestamp*/
        System.out.println(data.getTime()); // 1656335159459

        System.out.println(data.toString()); // Mon Jun 27 15:01:09 CEST 2022
    }
}
```

La classe Date rappresenta un intervallo di tempo che va dal 01/01/1970 all'istante in cui creiamo l'oggetto di tale classe.

JAVA.UTIL.CALENDAR E JAVA.UTIL.GREGORIANCALENDAR

Calendar e GregorianCalendar sono delle classi introdotte nella versione 2 di JAVA.

La classe Calendar è una classe astratta, che definisce (senza implementarli) tutti i metodi per la gestione e la manipolazione delle date.

La classe GregorianCalendar è un'implementazione della classe Calendar, che appunto implementa tutti i metodi definiti.

Per ottenere un'istanza della classe Calendar, dobbiamo utilizzare il metodo statico getInstance() della classe Calendar, oppure creare semplicemente un new GregorianCalendar();

Esempio

```
Calendar data = Calendar.getInstance();

Calendar data2 = new GregorianCalendar();
```

Vediamo un esempio:

```
import java.util.Calendar;
import java.util.Date;
import java.util.GregorianCalendar;

public class EsempioCalendar {

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

```

EsempioCalendar esempioCalendar = new EsempioCalendar();

esempioCalendar.esCalendar();
}

private void esCalendar() {
    Calendar cal = Calendar.getInstance();

    Calendar cal2 = new GregorianCalendar();

    /* setTime() imposta come data del Calendar, l'istante in cui creiamo l'istanza della classe Date */
    cal.setTime(new Date());

    /* il metodo set, che prende in ingresso due parametri, ci consente di specificare mediante il primo
    parametro cosa vogliamo settare (giorno, mese, anno, minuto, secondo), mentre mediante il secondo
    parametro il valore */
    cal2.set(Calendar.YEAR, 2000);
    cal2.set(Calendar.MONTH, 10); /* I mesi vanno da 0 a 11*/
    cal2.set(Calendar.DATE, 29);

    System.out.println(cal2.getTime()); // Wed Nov 29 15:54:40 CET 2000

    /* get() permette di recuperare il pezzo di data richiesto*/
    System.out.println(cal2.get(Calendar.YEAR)); // 2000
}
}

```

JAVA.TIME.LOCALDATE

La classe LocalDate è una delle classi più importanti ed è una classe immutabile, come la classe String.

Immutabile vuol dire che non può essere ereditata, quindi non si può estendere.

Un'istanza di LocalDate rappresenta un valore senza alcuna informazione temporale.

Ovviamente, la data impostata ha inizio in momenti diversi nella timeline, in base alla posizione sulla terra del dispositivo su cui viene creata l'istanza (ad esempio, a Roma la data locale inizierà 6 ore prima di New York e 9 ore prima di San Francisco).

Quindi, quello che ci interessa sapere è che la classe LocalDate rappresenta un valore localizzato, sulla base della posizione del dispositivo su cui viene creata una sua istanza.

Per creare un'istanza della classe LocalDate, è necessario utilizzare uno dei metodi statici messi a disposizione all'interno della classe:

- LocalDate.of(anno, mese, giorno);
- LocalDate.from(altri oggetti temporali).

Esempi di creazione di un'istanza della classe LocalDate

```

LocalDate ldA = LocalDate.of(2017, Month.MAY, 31);
LocalDate ldB = LocalDate.from(ldA);

```

Vediamo alcuni metodi utili della classe LocalDate:

- `isLeapYear()`: ritorna true se l'anno è bisestile, false altrimenti;
- `lengthOfMonth()`: ritorna il numero dei giorni di un mese;
- `getDayOfWeek()`: ritorna il giorno della settimana;
- `withYear(int year)`, `withMonth(int month)`, `withDayOfMonth(int dayOfMonth)`: ritornano un oggetto LocalDate con la data modificata rispettivamente nell'anno, nel mese e nel giorno rispetto la data di un altro oggetto LocalDate;
- `plusYears(long yearsToAdd)`, `plusMonths(long monthsToAdd)`, `plusDays(long daysToAdd)`: ritornano un oggetto LocalDate con la data aumentata rispettivamente nell'anno, nel mese e nel giorno rispetto la data di un altro oggetto LocalDate;
- `minusYears(long yearsToSubtract)`, `minusMonths(long monthsToSubtract)`, `minusDays(long daysToSubtract)`: ritornano un oggetto LocalDate con la data diminuita rispettivamente nell'anno, nel mese e nel giorno rispetto la data di un altro oggetto LocalDate;

Vediamo un esempio:

```
import java.time.LocalDate;

public class EsempioLocalDate {
    public static void main(String[] args) {
        EsempioLocalDate esempioLocalDate = new EsempioLocalDate();

        esempioLocalDate.esLocalDate();
    }

    private void esLocalDate() {
        LocalDate localDate = LocalDate.of(2017, 3, 29);

        System.out.println(localDate.isLeapYear()); // false
        System.out.println(localDate.lengthOfMonth()); // 31
        System.out.println(localDate.getDayOfWeek()); // WEDNESDAY

        LocalDate localDate2 = LocalDate.from(localDate);
        localDate2 = localDate.withYear(2016);
        System.out.println(localDate2); // 2017-03-29

        LocalDate localDate3 = LocalDate.from(localDate);
        localDate3 = localDate.plusMonths(2); // 2017-05-29
        System.out.println(localDate3);

        LocalDate localDate4 = LocalDate.from(localDate);
        localDate4 = localDate.minusDays(5); // 2017-03-24
        System.out.println(localDate4);
    }
}
```

JAVA.TIME.LOCALTIME

La classe LocalTime è utilizzata per la gestione delle ore.

Il classico esempio è quello di gestire le ore di un esercizio commerciale.

Un'istanza della classe LocalTime rappresenta un orario, senza una data associata e senza un fuso orario.

Esempi di creazione di un'istanza della classe LocalTime

```
LocalTime timeA = LocalTime.of(9, 0);
```

Vediamo alcuni metodi utili della classe LocalDate:

- `getHour()`: ritorna l'ora impostata;
- `getMinute()`: ritorna i minuti;
- `withHour(int hour)`, `withMinute(int minute)`, `withSecond(int second)`: ritornano un oggetto LocalTime con l'orario modificato rispettivamente nell'ora, nei minuti e nei secondi rispetto l'orario di un altro oggetto LocalTime;
- `plusHours(long hoursToAdd)`, `plusMinutes(int minutesToAdd)`, `plusSeconds(int secondsToAdd)`: ritornano un oggetto LocalTime con l'orario aumentato rispettivamente nell'ora, nei minuti e nei secondi rispetto l'orario di un altro oggetto LocalTime;
- `minusHours(long hoursToSubtract)`, `minusMinutes(int minutesToSubtract)`, `plusSeconds(int secondsToSubtract)`: ritornano un oggetto LocalTime con l'orario diminuito rispettivamente nell'ora, nei minuti e nei secondi rispetto l'orario di un altro oggetto LocalTime;

Vediamo un esempio:

```
import java.time.LocalTime;

public class EsempioLocalTime {
    public static void main(String[] args) {
        EsempioLocalTime esempioLocalTime = new EsempioLocalTime();

        esempioLocalTime.esLocalTime();
    }

    private void esLocalTime() {
        LocalTime localTime = LocalTime.of(9, 30);

        System.out.println(localTime); // 09:30
        System.out.println(localTime.getHour()); // 9
        System.out.println(localTime.getMinute()); // 30

        LocalTime localTime2 = LocalTime.from(localTime);
        localTime2 = localTime.withSecond(10);
        System.out.println(localTime2); // 09:30:10

        LocalTime localTime3 = LocalTime.from(localTime);
        localTime3 = localTime.plusMinutes(3);
        System.out.println(localTime3); // 09:33
    }
}
```

```

        LocalTime localTime4 = LocalTime.from(localTime);
        localTime4 = localTime.minusHours(3);
        System.out.println(localTime4); // 06:30
    }
}

```

JAVA.TIME.LOCALDATETIME

Questa classe consente di gestire data e ora in maniera congiunta.

Un'istanza della classe LocalDateTime rappresenta sia una data che un orario.

JAVA.TIME.PERIOD E JAVA.TIME.DURATION

Le classi Period e Duration vengono utilizzate per rappresentare dei periodi di tempo (ad esempio 3 mesi e 5 giorni).

La classe Duration consente di rappresentare un intervallo di tempo in secondi e nanosecondi (ad esempio 50,3 secondi).

La classe Period consente invece di rappresentare un intervallo di tempo in anni, mesi e giorni (ad esempio 2 anni 3 mesi e 5 giorni).

Queste classi le possiamo utilizzare per sommare o sottrarre periodi dalle classi Date e Time.

Esempi di utilizzo

```

LocalDate ldA = LocalDate.of(2017, Month.MAY, 31);
LocalDate ldB = LocalDate.from(ldA);

```

Duration.ofSeconds(3, 10); // ritorna un'istanza della classe Duration che rappresenta un intervallo di tempo di 3 secondi e 10 nanosecondi

Duration.between(IdA, IdB); // ritorna un'istanza della classe Duration che contiene l'intervallo di tempo tra le due date IdA ed IdB

Period.ofMonths(4); // ritorna un'istanza della classe Period che rappresenta un intervallo di tempo di 4 mesi

Period.between(IdA, IdB); // ritorna un'istanza della classe Period che rappresenta l'intervallo di tempo che intercorre tra due date IdA ed IdB.

Vediamo un esempio:

```
import java.time.Duration;
import java.time.LocalTime;

public class EsempioPeriodDuration {
    public static void main(String[] args) {
        EsempioPeriodDuration esempioPeriodDuration = new EsempioPeriodDuration();

        esempioPeriodDuration.esPeriodDuration();
    }

    private void esPeriodDuration() {
        Duration d1 = Duration.ofSeconds(10, 40);

        System.out.println(d1.getSeconds()); // 10
        System.out.println(d1.getNano()); // 40

        LocalTime aperturaNegozio = LocalTime.of(9, 30);
        LocalTime chiusuraNegozio = LocalTime.of(13, 30);

        Duration d2 = Duration.between(aperturaNegozio, chiusuraNegozio);
        System.out.println(d2.getSeconds()); // 14400
        System.out.println(d2.getNano()); // 0

        /* Analogamente posso usare il Period, però manipolando gli oggetti LocalDate
        */
    }
}
```

JAVA.TEXT.SIMPLDATEFORMAT

La classe `SimpleDateFormat` viene in aiuto per gestire la formattazione delle date.

Questa classe prende in ingresso nel costruttore il formato della data e, data una stringa e mediante il metodo `parse()`, ritorna un oggetto di tipo `Date` con la data formattata secondo il template del costruttore.

Esempio

```
public static void main(String[] args) {
    Calendar data = Calendar.getInstance();

    SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy - HH:mm:ss");
    SimpleDateFormat sdf2 = new SimpleDateFormat("yyyy/MM/dd - HH:mm:ss");
    SimpleDateFormat sdf3 = new SimpleDateFormat("dd/MM/yyyy");

    System.out.println("Oggi è: " + sdf.format(data.getTime()));
    System.out.println("Oggi è: " + sdf2.format(data.getTime()));
    System.out.println("Oggi è: " + sdf3.format(data.getTime()));

    SimpleDateFormat sdf4 = new SimpleDateFormat("dd/MM/yyyy");
    Date d4 = sdf4.parse("20/02/2016");
    Calendar c4 = Calendar.getInstance();
    c4.setTime(d4);
    System.out.println("La data impostata è: " + c4.getTime());
}
```

Il costruttore `SimpleDateFormat` accetta in ingresso una stringa contenente il formato in cui deve essere visualizzata la data.

Per ottenere una data formattata secondo le nostre esigenze dobbiamo utilizzare il metodo **format(Date date)**.

Se vogliamo convertire una stringa in un oggetto `Date` dobbiamo utilizzare il metodo `parse(String source)`.

ATTENZIONE: la data deve essere scritta nel formato impostato nel costruttore di `SimpleDateFormat` altrimenti verrà generata l'eccezione `ParseException`

Vediamo un esempio:

```
import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.GregorianCalendar;

public class EsempioSimpleDateFormat {
    public static void main(String[] args) {
        EsempioSimpleDateFormat esempioSimpleDateFormat = new
EsempioSimpleDateFormat();

        esempioSimpleDateFormat.esSimpleDateFormat();
    }

    private void esSimpleDateFormat() {
        Calendar c = new GregorianCalendar();
        c.set(Calendar.YEAR, 2017);
        c.set(Calendar.MONTH, 4);
        c.set(Calendar.DATE, 29);
        c.set(Calendar.HOUR, 10);
        c.set(Calendar.MINUTE, 30);
        c.set(Calendar.SECOND, 25);

        SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");
        String data = sdf.format(c.getTime());
        System.out.println(data); // 29/05/2017

        SimpleDateFormat sdf2 = new SimpleDateFormat("dd/MM/yyyy - HH:mm:ss");
        String data2 = sdf2.format(c.getTime());
        System.out.println(data2); // 29/05/2017 - 22:30:25
    }
}
```

DATABASE

INTRODUZIONE AI DATABASE

Un database è un software che consente di gestire una grande quantità di dati, il loro salvataggio, il loro recupero, aggiornamenti e così via.

Immaginando di dover gestire un software gestionale o un sistema di messaggistica, avremo bisogno di gestire i dati che vengono inseriti o richiesti dall'utente. Se non esistessero i database, dovremmo gestire in maniera manuale tutti questi dati attraverso dei file, con notevole dispendio di tempo, di risorse e scarsi risultati.

I database nascono appunto per gestire tutti i dati che possono far parte di un software. A seconda del software, chiaramente avremo più o meno dati.

Esistono diverse tipologie di database:

-RDBMS (Relational DataBase Management System);

-NoSQL Database;

Nell'ambito dei database relazionali abbiamo diverse piattaforme, open source o a pagamento.

Ogni database consente di gestire i dati organizzati in tabelle provviste di colonne di vario tipo e, attraverso un linguaggio chiamato SQL, è possibile effettuare delle ricerche, oppure inserire dati, al loro interno.

OPERAZIONI CRUD

CRUD sta per CREATE READ UPDATE DELETE, ovvero inserire righe nelle tabelle, leggerle, aggiornarle e cancellarle. Vediamo di seguito la sintassi in SQL per svolgere queste 4 operazioni:

- CREATE:

```
INSERT INTO nome_database.nome_tabella (campo1, campo2, ... campon) VALUES (valore1,  
valore2, ... valoren );
```

Per esempio:

```
INSERT INTO corso_java.clienti (nome, cognome, email, telefono) VALUES ('Laura', 'Martinelli',  
'test@marti.it', '1111111');
```

- READ:

```
SELECT campo1, campo2, ... campon FROM nome_database.nome_tabella;  
SELECT * FROM nome_database.nome_tabella;  
SELECT * FROM nome_database.nome_tabella WHERE condizioni...  
    campo1 di tipo BIGINT -> WHERE campo1 = 30 oppure WHERE campo1 IN(30, 35, 37, 40,  
    343245)  
    campo2 di tipo VARCHAR -> WHERE campo2 = 'teSTo' oppure WHERE campo2 LIKE '%testo%'  
    oppure LOWER(campo2) = LIKE(LOWER("%teSTo%")) oppure UPPER(campo2) = LIKE  
    (UPPER('%test'))
```

```
WHERE campo2 LIKE '%testo%'  
WHERE campo2 LIKE 'testo%'  
WHERE campo2 LIKE '%testo'
```

Per esempio:

```
SELECT * FROM corso_java.clienti WHERE nome = 'Paolo';  
  
/*Questa query restituisce tutte le righe in cui il nome contiene 'Pao'*/  
SELECT * FROM corso_java.clienti WHERE nome LIKE '%Pao%';  
  
/*Questa query restituisce tutte le righe in cui il nome inizia per 'P'*/  
SELECT * FROM corso_java.clienti WHERE nome LIKE 'P%';  
  
/*Questa query restituisce tutte le righe in cui il nome finisce per 'o'*/  
SELECT * FROM corso_java.clienti WHERE nome LIKE 'o%';  
  
LIMIT indica il numero massimo di righe da recuperare  
      SELECT * FROM nome_database.nome_tabella LIMIT 30 -> la query tornerà al massimo 30 righe
```

ORDER BY indica l'ordine di visualizzazione delle righe: ASC (crescente) o DESC (decrescente)
 SELECT * FROM nome_database.nome_tabella ORDER BY campon ASC

```
SELECT  
      nome_database.nome_tabella1.campo1, nome_database.nome_tabella2.campo1  
FROM  
      nome_database.nome_tabella1, nome_database.nome_tabella2  
WHERE  
      nome_database.nome_tabella1.campoX = nome_database.nome_tabella2.campoY
```

```
SELECT campo1 AS nome, campo2 AS cognome, ... campon FROM nome_database.nome_tabella;
```

Esempio:

```
SELECT c.cognome, c.nome, o.importo, o.id AS 'NUMERO ORDINE'  
FROM corso_java.clienti AS c, corso_java.ordini AS o  
WHERE o.id_cliente = c.id;
```

```
SELECT  
      a.campo1, b.campo1  
FROM  
      nome_database.nome_tabella1 AS a, nome_database.nome_tabella2 AS b  
WHERE  
      a.campoX = b.campoY
```

- UPDATE:

```
UPDATE nome_database.nome_tabella SET campo1 = valore 1, campo2 = valore2, ... campom =
```

valorem (m <= n)

Esempio

```
UPDATE corso_java.clienti SET telefono = '44444444';
```

- DELETE

```
DELETE FROM nome_database.nome_tabella /* cancello tutte le righe */  
DELETE FROM nome_database.nome_tabella WHERE condizioni...
```

INTERFACCIARSI CON I DATABASE IN JAVA – JDBC

Prima di interfacciarsi con il database, dobbiamo scaricare il JDBC connector.

Il JDBC connector è un jar che contiene delle librerie, che consentono di stabilire la connessione con un database. Esistono connector per vari tipi di database, ma nel caso nostro scaricheremo il JDBC connector per MySql mediante il sito ufficiale di MySql (<https://dev.mysql.com/downloads/connector/j/?os=26>).

Dopo aver trovato e scaricato il JDBC connector in formato zip, estraendolo possiamo trovarci all'interno il jar corrispondente. Tale jar va inserito all'interno del progetto come dipendenza, ma la procedura per eseguire questo inserimento cambia in base all'IDE, quindi ci si deve informare su come fare prima di andare avanti.

Una volta effettuato l'inserimento del JDBC nel progetto, vediamo di seguito un esempio di connessione al database:

```
import com.mysql.cj.jdbc.MysqlDataSource;  
  
import java.sql.Connection;  
import java.sql.SQLException;  
  
public class EsempioConnessioneDatabase {  
  
    /* Il dump del database esportato da MySql è nella cartella db del progetto Corso_JAVA_EE */  
  
    private Connection con;  
  
    public static void main(String[] args) {  
        EsempioConnessioneDatabase esempioConnessioneDatabase = new EsempioConnessioneDatabase();  
  
        try {  
            /* Stampa false se la connessione è aperta, quindi se ha avuto successo */  
            System.out.println(esempioConnessioneDatabase.getConnection().isClosed()); // false  
        } catch (SQLException e) {  
            e.printStackTrace();  
        }  
    }  
  
    private Connection getConnection() throws SQLException {  
        /* Se l'istanza Connection è null, la definiamo noi con la classe MysqlDataSource*/  
        if (con == null) {  
            MysqlDataSource dataSource = new MysqlDataSource();  
  
            /* Indica indirizzo ip dove risiede il database*/  
            dataSource.setServerName("localhost");  
        }  
        return con;  
    }  
}
```

```

        dataSource.setPortNumber(3306);

        dataSource.setUser("root");
        dataSource.setPassword("root");

        dataSource.setDatabaseName("corso_java");

        /* Se tutto è andato per il meglio,
         * getConnection() mi restituisce un'istanza della connessione*/
        con = dataSource.getConnection();
    }

    return con;
}
}

```

OPERAZIONI CRUD CON IL JDBC CONNECTOR

SELECT CON JDBC

Una volta che siamo riusciti a connetterci al database mediante il JDBC connector, vediamo ora come possiamo effettuare le ricerche all'interno di tale database, mediante il comando SELECT.

Di seguito un esempio:

```

import com.mysql.cj.jdbc.MysqlDataSource;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class EsempioConnessioneDatabase {

    /* Il dump del database esportato da MySQL è nella cartella db del progetto Corso_JAVA_EE */

    private Connection con;

    public static void main(String[] args) {
        EsempioConnessioneDatabase esempioConnessioneDatabase = new EsempioConnessioneDatabase();

        try {
            /* Stampa false se la connessione è aperta, quindi se ha avuto successo */
            System.out.print("La connessione al database e' fallita: ");
            System.out.println(esempioConnessioneDatabase.getConnection().isClosed()); // false
            System.out.println("-----");

            System.out.println("Le righe della query 'SELECT id, nome, cognome, email, telefono FROM clienti' sono le seguenti: ");
            System.out.println("-----");
            esempioConnessioneDatabase.esempioSelect();

            System.out.println("Le righe della query 'SELECT id, nome, cognome, email, telefono FROM clienti WHERE cognome LIKE '%Pre%' sono le seguenti: ");
            System.out.println("-----");
            esempioConnessioneDatabase.esempioSelect2();

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    private Connection getConnection() throws SQLException {
        /* Se l'istanza Connection è null, la definiamo noi con la classe MysqlDataSource*/
        if (con == null) {

```

```

MysqlDataSource dataSource = new MysqlDataSource();

/* Indica indirizzo ip dove risiede il database*/
dataSource.setServerName("localhost");

dataSource.setPortNumber(3306);

dataSource.setUser("root");
dataSource.setPassword("root");

dataSource.setDatabaseName("corso_java");

/* Se tutto è andato per il meglio,
* getConnection() mi restituisce un'istanza della connessione*/
con = dataSource.getConnection();
}

return con;
}

private void esempioSelect() throws SQLException {
/* Definiamo la query che restituisce l'elenco di tutti i clienti */
String sql = "SELECT id, nome, cognome, email, telefono FROM clienti" ;

/* Per eseguire la query, serve un oggetto di tipo PreparedStatement.
Tale oggetto serve per passare in input a MySQL la query, in modo da
poterla effettuare. */
PreparedStatement preparedStatement = getConnection().prepareStatement(sql);

/* Il risultato delle query è un oggetto di tipo ResultSet */
ResultSet resultSet = preparedStatement.executeQuery();

/* Mediante il seguente while, estrapoliamo tutte le righe trovate dalla query,
per poi stamparle */
while(resultSet.next()) {

/* Accediamo alle colonne dell' i-esima riga */
System.out.println("id = " + resultSet.getInt(1));
System.out.println("nome = " + resultSet.getString(2));
System.out.println("cognome = " + resultSet.getString(3));
System.out.println("email = " + resultSet.getString(4));
System.out.println("telefono = " + resultSet.getString(5));
System.out.println("-----");
}
}

private void esempioSelect2() throws SQLException {
/* Definiamo la query che restituisce l'elenco di tutti i clienti */
String sql = "SELECT id, nome, cognome, email, telefono FROM clienti WHERE cognome LIKE '%Pre%' " ;

/* Per eseguire la query, serve un oggetto di tipo PreparedStatement.
Tale oggetto serve per passare in input a MySQL la query, in modo da
poterla effettuare. */
PreparedStatement preparedStatement = getConnection().prepareStatement(sql);

/* Il risultato delle query è un oggetto di tipo ResultSet */
ResultSet resultSet = preparedStatement.executeQuery();

/* Mediante il seguente while, estrapoliamo tutte le righe trovate dalla query,
per poi stamparle */
while(resultSet.next()) {

/* Accediamo alle colonne dell' i-esima riga */
System.out.println("id = " + resultSet.getInt(1));
System.out.println("nome = " + resultSet.getString(2));
System.out.println("cognome = " + resultSet.getString(3));
System.out.println("email = " + resultSet.getString(4));
System.out.println("telefono = " + resultSet.getString(5));
System.out.println("-----");
}
}
}

```

L'output di tale esempio è il seguente:

```
La connessione al database e' fallita: false
-----
Le righe della query 'SELECT id, nome, cognome, email, telefono FROM clienti' sono le seguenti:
-----
id = 1
nome = Paolo
cognome = Preite
email = info@paolopreite.it
telefono = 1111111
-----
id = 2
nome = Chiara
cognome = Martini
email = test@test.it
telefono = 2222222
-----
id = 3
nome = Laura
cognome = Martinelli
email = test@marti.it
telefono = 1111111
-----
Le righe della query 'SELECT id, nome, cognome, email, telefono FROM clienti WHERE cognome LIKE
'%Pre%' sono le seguenti:
-----
id = 1
nome = Paolo
cognome = Preite
email = info@paolopreite.it
telefono = 1111111
-----
```

INSERT, UPDATE E DELETE CON JDBC

Ora vediamo l'interazione con il database con il JDBC connector in scrittura, aggiornamento e cancellazione, mediante i comandi INSERT, UPDATE E DELETE.

Di seguito un esempio:

```
import com.mysql.cj.jdbc.MysqlDataSource;
import java.sql.*;
public class EsempioConnessioneDatabase {
    /* IL dump del database esportato da MySQL è nella cartella db del progetto Corso_JAVA_EE */
    private Connection con;
    public static void main(String[] args) {
        EsempioConnessioneDatabase esempioConnessioneDatabase = new EsempioConnessioneDatabase();
        try {
            /* Stampa false se la connessione è aperta, quindi se ha avuto successo */
            System.out.print("La connessione al database e' fallita: ");
            System.out.println(esempioConnessioneDatabase.getConnection().isClosed()); // false
            System.out.println("-----");
            System.out.println("Le righe della query 'SELECT id, nome, cognome, email, telefono FROM clienti' sono le seguenti: ");
            System.out.println("-----");
            esempioConnessioneDatabase.esempioSelect();
```

```

        System.out.println("Le righe della query 'SELECT id, nome, cognome, email, telefono FROM clienti
WHERE cognome LIKE '%Pre%' sono le seguenti:");
        System.out.println("-----");
        esempioConnessioneDatabase.esempioSelect2();

        esempioConnessioneDatabase.esempioInsert("Sara", "Bartoli", "sara@test.it", "9999999");
        System.out.println("-----");

        esempioConnessioneDatabase.esempioUpdate();

        esempioConnessioneDatabase.esempioDelete();

    } catch (SQLException e) {
        e.printStackTrace();
    }
}

private Connection getConnection() throws SQLException {
    /* Se l'istanza Connection è null, la definiamo noi con la classe MysqlDataSource*/
    if (con == null) {
        MysqlDataSource dataSource = new MysqlDataSource();

        /* Indica indirizzo ip dove risiede il database*/
        dataSource.setServerName("localhost");

        dataSource.setPortNumber(3306);

        dataSource.setUser("root");
        dataSource.setPassword("root");

        dataSource.setDatabaseName("corso_java");

        /* Se tutto è andato per il meglio,
        * getConnection() mi restituisce un'istanza della connessione*/
        con = dataSource.getConnection();
    }

    return con;
}

private void esempioSelect() throws SQLException {
    /* Definiamo La query che restituisce l'elenco di tutti i clienti */
    String sql = "SELECT id, nome, cognome, email, telefono FROM clienti" ;

    /* Per eseguire la query, serve un oggetto di tipo PreparedStatement.
    Tale oggetto serve per passare in input a MySql la query, in modo da
    poterla effettuare. */
    PreparedStatement preparedStatement = getConnection().prepareStatement(sql);

    /* Il risultato delle query è un oggetto di tipo ResultSet */
    ResultSet resultSet = preparedStatement.executeQuery();

    /* Mediante il seguente while, estrapoliamo tutte le righe trovate dalla query,
    per poi stamparle */
    while(resultSet.next()) {

        /* Accediamo alle colonne dell' i-esima riga */
        System.out.println("id = " + resultSet.getInt(1));
        System.out.println("nome = " + resultSet.getString(2));
        System.out.println("cognome = " + resultSet.getString(3));
        System.out.println("email = " + resultSet.getString(4));
        System.out.println("telefono = " + resultSet.getString(5));
        System.out.println("-----");
    }
}

private void esempioSelect2() throws SQLException {
    /* Definiamo La query che restituisce l'elenco di tutti i clienti */
    String sql = "SELECT id, nome, cognome, email, telefono FROM clienti WHERE cognome LIKE '%Pre%' " ;

    /* Per eseguire la query, serve un oggetto di tipo PreparedStatement.
    Tale oggetto serve per passare in input a MySql la query, in modo da
    poterla effettuare. */
    PreparedStatement preparedStatement = getConnection().prepareStatement(sql);
}

```

```

/* Il risultato delle query è un oggetto di tipo ResultSet */
ResultSet resultSet = preparedStatement.executeQuery();

/* Mediante il seguente while, estrapoliamo tutte le righe trovate dalla query,
per poi stamparle */
while(resultSet.next()) {

    /* Accediamo alle colonne dell' i-esima riga */
    System.out.println("id = " + resultSet.getInt(1));
    System.out.println("nome = " + resultSet.getString(2));
    System.out.println("cognome = " + resultSet.getString(3));
    System.out.println("email = " + resultSet.getString(4));
    System.out.println("telefono = " + resultSet.getString(5));
    System.out.println("-----");
}
}

private void esempioInsert(String nome, String cognome, String email, String telefono) throws
SQLException {
    String sql = "INSERT INTO clienti(nome, cognome, email, telefono) " +
        "VALUES ('"+nome+"', '"+cognome+"', '"+email+"', '"+telefono+"')";

    /* IL RETURN_GENERATED_KEYS è da utilizzare nei casi in cui
    l'id è generato automaticamente dal database */
    PreparedStatement preparedStatement = getConnection().prepareStatement(sql,
Statement.RETURN_GENERATED_KEYS);

    preparedStatement.executeUpdate();

    ResultSet resultSet = preparedStatement.getGeneratedKeys();
    resultSet.next();

    System.out.println("L'id della nuova riga inserita (INSERT) e' " + resultSet.getInt(1));
}

private void esempioUpdate () throws SQLException {
    String sql = "UPDATE clienti SET telefono = '12345678' WHERE id = 5";

    PreparedStatement preparedStatement = getConnection().prepareStatement(sql,
Statement.RETURN_GENERATED_KEYS);

    preparedStatement.executeUpdate();
}

private void esempioDelete () throws SQLException {
    String sql = "DELETE FROM clienti WHERE nome = 'Sara'";

    PreparedStatement preparedStatement = getConnection().prepareStatement(sql,
Statement.RETURN_GENERATED_KEYS);

    preparedStatement.executeUpdate();
}
}

```

PREVENIRE ATTACCHI SQL INJECTION CON I PREPAREDSTATEMENT

Un attacco di tipo SQL injection è un attacco hacker molto particolare, e molto frequente se l'applicazione viene progettata male, che può causare notevoli danni all'applicazione stessa, compresa la cancellazione totale dei dati.

SQL injection vuol dire che quando eseguiamo una query, all'interno della stessa query è presente un'istruzione che effettua delle operazioni malevoli, ad esempio il DROP del database, la cancellazione di una tabella, il recupero di informazioni dal database e così via.

Quindi, bisogna stare attenti a progettare la classe che effettua le interrogazioni al database, per prevenire appunto attacchi di questo tipo.

In JAVA, lo strumento che viene messo a disposizione per prevenire attacchi di tipo SQL injection è il PreparedStatement, che abbiamo visto già in precedenza per effettuare le operazioni CRUD, ma che non abbiamo utilizzato ancora nella maniera corretta per prevenire tali attacchi. Infatti, passare direttamente alla stringa, ovvero alla query, i parametri che ne determineranno il risultato, permette agli utenti malevoli di effettuare attacchi hacker molto più facilmente, perché inserirebbero l'istruzione di tipo SQL injection proprio in questi parametri.

Vediamo quindi di seguito il modo corretto di utilizzare il PreparedStatement, senza passare parametri in ingresso alla query:

```
import com.mysql.cj.jdbc.MysqlDataSource;  
  
import java.sql.*;  
  
public class EsempioUtilizzoPreparedStatementPerPrevenireAttacchiSQLInjection {  
    /* Il dump del database esportato da MySQL è nella cartella db del progetto Corso_JAVA_EE */  
  
    private Connection con;  
  
    public static void main(String[] args) {  
        database.EsempioUtilizzoPreparedStatementPerPrevenireAttacchiSQLInjection esPreparedStatement = new  
        database.EsempioUtilizzoPreparedStatementPerPrevenireAttacchiSQLInjection();  
  
        try {  
            /* Stampa false se la connessione è aperta, quindi se ha avuto successo */  
            System.out.print("La connessione al database è fallita: ");  
            System.out.println(esPreparedStatement.getConnection().isClosed()); // false  
            System.out.println("-----");  
  
            String cognome = "Preite";  
            System.out.println("Le righe della query \"SELECT id, nome, cognome, email, telefono FROM clienti  
WHERE cognome = " + "''" + cognome + "''" + "\" sono le seguenti: ");  
            System.out.println("-----");  
            esPreparedStatement.esempioSelect(cognome);  
  
            esPreparedStatement.esempioInsert("Giorgia", "Bartoli", "sara@test.it", "9999999");  
            System.out.println("-----");  
  
            esPreparedStatement.esempioUpdate("54321", 2);  
  
            esPreparedStatement.esempioDelete("Sara");  
        } catch (SQLException e) {  
            e.printStackTrace();  
        }  
    }  
  
    private Connection getConnection() throws SQLException {  
        /* Se l'istanza Connection è null, la definiamo noi con la classe MysqlDataSource*/  
        if (con == null) {  
            MysqlDataSource dataSource = new MysqlDataSource();  
        }  
        return con;  
    }  
}
```

```

/* Indica indirizzo ip dove risiede il database*/
dataSource.setServerName("localhost");

dataSource.setPortNumber(3306);

dataSource.setUser("root");
dataSource.setPassword("root");

dataSource.setDatabaseName("corso_java");

/* Se tutto è andato per il meglio,
 * getConnection() mi restituisce un'istanza della connessione*/
con = dataSource.getConnection();
}

return con;
}

private void esempioSelect(String cognome) throws SQLException {
/* Definiamo La query che restituisce L'elenco di tutti i clienti */
String sql = "SELECT id, nome, cognome, email, telefono FROM clienti WHERE cognome = ?" ;

/* Per eseguire la query, serve un oggetto di tipo PreparedStatement.
Tale oggetto serve per passare in input a MySQL la query, in modo da
poterla effettuare. */
PreparedStatement preparedStatement = getConnection().prepareStatement(sql);
preparedStatement.setString(1, cognome);

/* Il risultato delle query è un oggetto di tipo ResultSet */
ResultSet resultSet = preparedStatement.executeQuery();

/* Mediante il seguente while, estrapoliamo tutte le righe trovate dalla query,
per poi stamparle */
while(resultSet.next()) {

/* Accediamo alle colonne dell' i-esima riga */
System.out.println("id = " + resultSet.getInt(1));
System.out.println("nome = " + resultSet.getString(2));
System.out.println("cognome = " + resultSet.getString(3));
System.out.println("email = " + resultSet.getString(4));
System.out.println("telefono = " + resultSet.getString(5));
System.out.println("-----");
}
}

private void esempioInsert(String nome, String cognome, String email, String telefono) throws SQLException {
String sql = "INSERT INTO clienti(nome, cognome, email, telefono) " +
"VALUES (?, ?, ?, ?)";

/* IL RETURN_GENERATED_KEYS è da utilizzare nei casi in cui
l'id è generato automaticamente dal database */
PreparedStatement preparedStatement = getConnection().prepareStatement(sql,
Statement.RETURN_GENERATED_KEYS);
preparedStatement.setString(1, nome);
preparedStatement.setString(2, cognome);
preparedStatement.setString(3, email);
preparedStatement.setString(4, telefono);

preparedStatement.executeUpdate();

ResultSet resultSet = preparedStatement.getGeneratedKeys();
resultSet.next();

System.out.println("L'id della nuova riga inserita (INSERT) e' " + resultSet.getInt(1));
}

private void esempioUpdate (String telefono, int id) throws SQLException {
String sql = "UPDATE clienti SET telefono = ? WHERE id = ?";

PreparedStatement preparedStatement = getConnection().prepareStatement(sql,
Statement.RETURN_GENERATED_KEYS);
preparedStatement.setString(1, telefono);
}

```

```
        preparedStatement.setInt(2, id);
        preparedStatement.executeUpdate();
    }

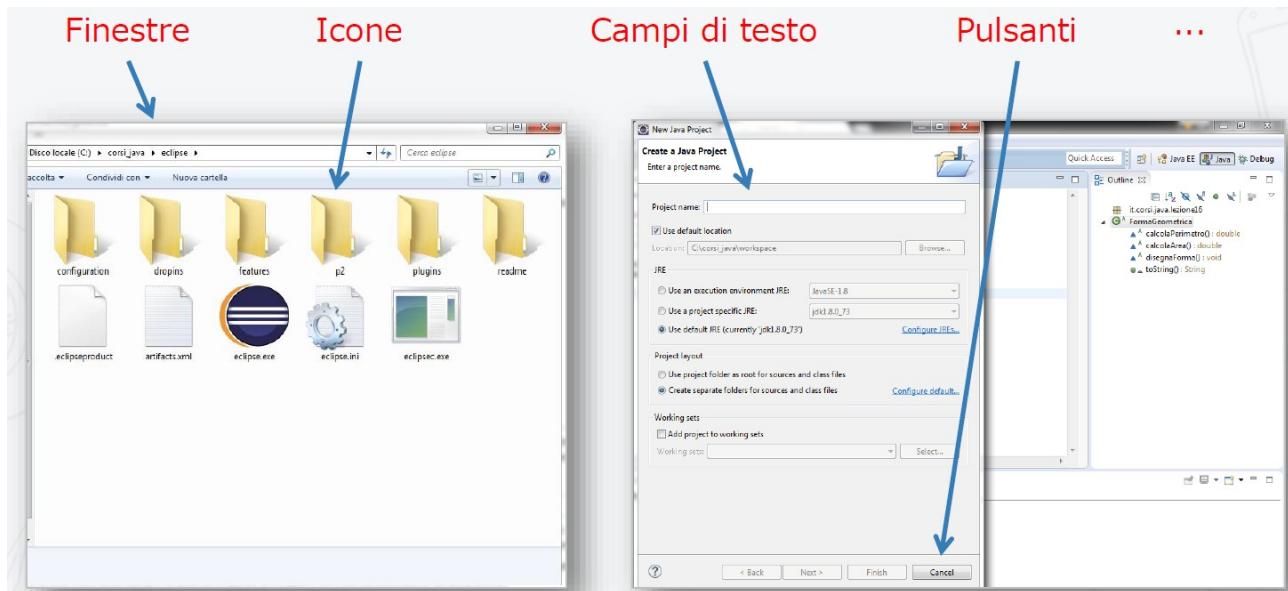
private void esempioDelete (String nome) throws SQLException {
    String sql = "DELETE FROM clienti WHERE nome = ?";
    PreparedStatement preparedStatement = getConnection().prepareStatement(sql,
Statement.RETURN_GENERATED_KEYS);
    preparedStatement.setString(1, nome);
    preparedStatement.executeUpdate();
}
```

LIBRERIE GRAFICHE IN JAVA

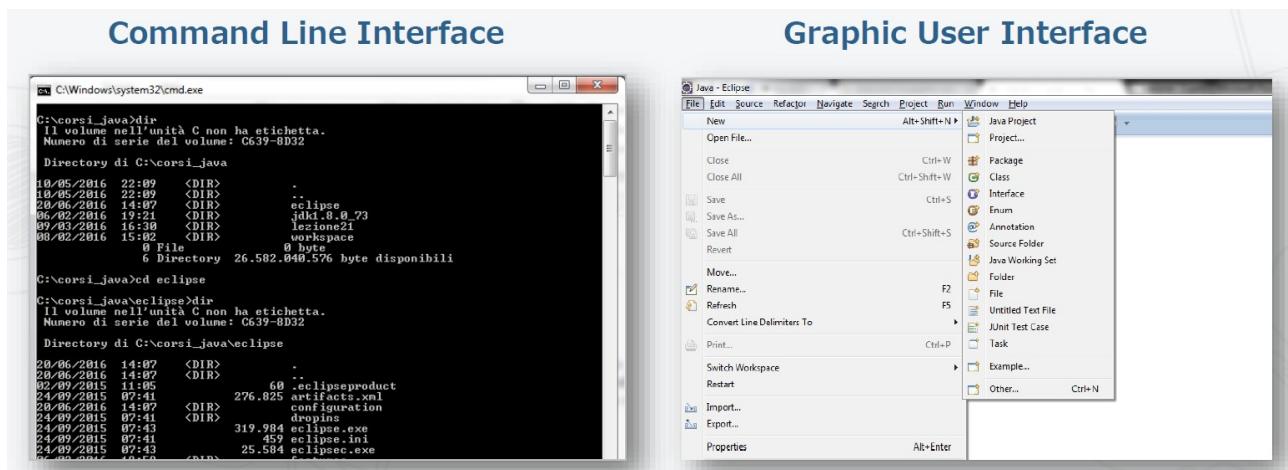
COMPONENTE GUI

GUI significa Graphical User Interface (interfaccia grafica), ed è l'insieme delle componenti grafiche che utilizziamo per interagire con il computer.

Quindi le icone, le finestre, le toolbar, i campi di testo e i pulsanti sono tutte componenti grafiche.



Prima della nascita delle GUI, esistevano le CLI (Command Line Interface), ovvero interfacce che consentivano di eseguire software a riga di comando.



Un toolkit grafico è un insieme di componenti software, sviluppati per semplificare e rendere uniforme lo sviluppo di GUI.

In JAVA i toolkit grafici sono suddivisi in due gruppi:

- Heavy Weight, che usano i widget forniti dal sistema operativo, quindi l'aspetto delle finestre cambia in funzione del sistema operativo su cui viene eseguita l'applicazione;

- Light Weight, che implementano i propri widget, quindi l'aspetto dell'applicazione rimane invariato, indipendentemente dal sistema operativo su cui l'applicazione viene eseguita.

Per creare applicazioni dotate di interfaccia grafica in JAVA, abbiamo a disposizione 3 librerie:

- AWT
- SWT
- Swing

AWT e SWT sono Heavy Weight toolkit, mentre Swing è un Light Weight toolkit.

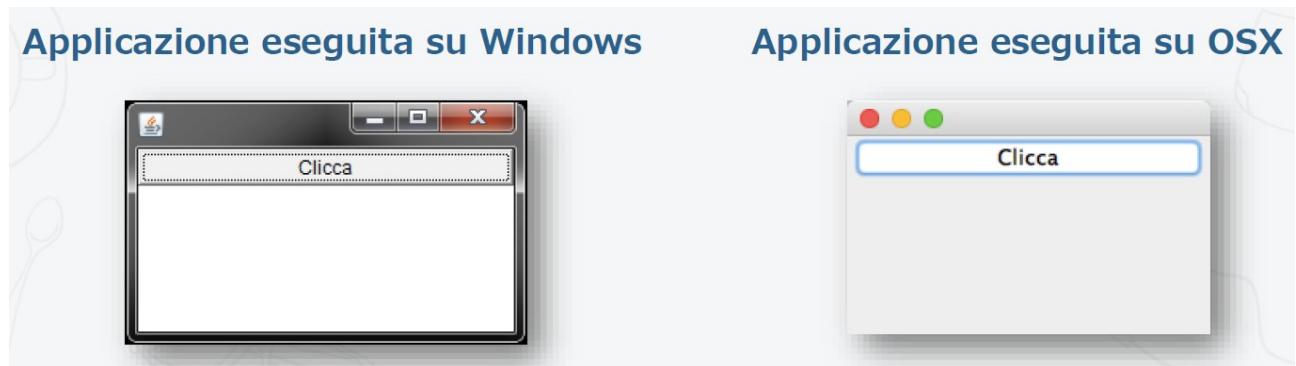
AWT

AWT sta per Abstract Window Toolkit, ed è la libreria Java che contiene le classi e le interfacce per il rendering grafico.

Attraverso AWT possiamo realizzare interfacce utente complesse e contiene:

- gli elementi di base (finestre, campi di testo, etichette, pulsanti, ...);
- l'interfaccia tra l'applicazione JAVA ed il sistema operativo;
- le librerie per la gestione del Drag and Drop;
- le interfacce per la gestione delle periferiche (mouse e tastiera);
- gli strumenti per l'esecuzione, da un'applicazione JAVA, di applicazioni del sistema operativo (browser, client di posta, ...).

Essendo AWT un Heavy Weight Toolkit, l'aspetto del software realizzato con questa libreria cambia a seconda del sistema operativo su cui viene eseguita.



Le classi per la creazione di una finestra sono Frame, Button e BorderLayout

```
import java.awt.BorderLayout;
import java.awt.Button;
import java.awt.Frame;

public class Finestra extends Frame {
    Button confirm = new Button("Clicca");

    public Finestra() {
        confirm.addActionListener(new FinestraListener());
        confirm.setActionCommand("CONFIRM");
        add(confirm, BorderLayout.NORTH);
        pack();
    }

    public static void main (String [] arg) {
        Finestra f = new Finestra();
        f.setVisible(true);
    }
}
```

```
import java.awt.Button;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class FinestraListener implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent e) {
        Button btn = (Button)e.getSource();

        if(btn.getLabel().equals("Clicca")) {
            btn.setLabel("Clicca 2");
        } else {
            btn.setLabel("Clicca");
        }
    }
}
```

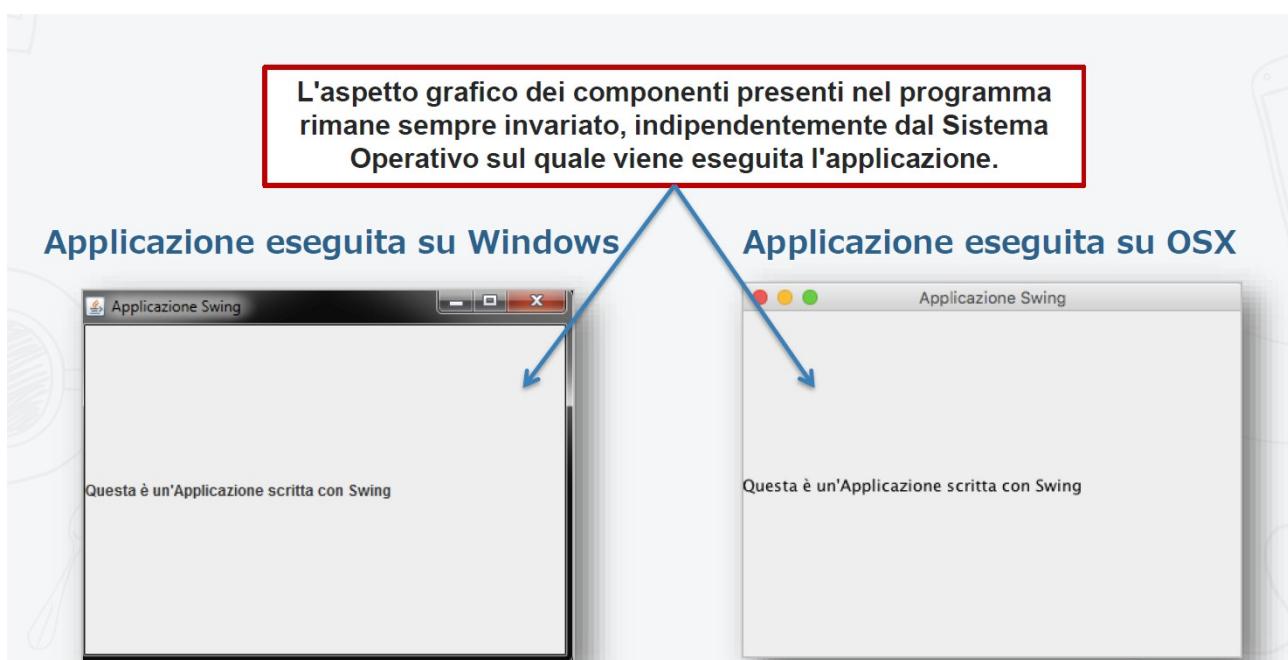
Nell'esempio viene creata una classe Finestra che estende Frame, viene creato il suo costruttore che aggiunge i vari componenti e viene creata una sua istanza per renderla visibile. All'interno del Listener, bisogna implementare il pulsante e l'azione su di esso.

SWING

Swing è un'estensione di AWT, e ha queste caratteristiche:

- è scritta interamente in JAVA;
- è indipendente dalla piattaforma;
- fornisce un set più ricco di componenti rispetto AWT;

Swing appartiene alla categoria dei Light Weight toolkit, quindi un'applicazione sviluppata in Swing avrà lo stesso aspetto indipendentemente dal sistema operativo sul quale viene eseguita.



Vediamo un esempio in codice:

```
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.SwingUtilities;
public class FinestraSwing {
    public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                mainOnEventDispatchThread();
            }
        });
    }
    private static void mainOnEventDispatchThread() {
        JFrame f = new JFrame("Applicazione Swing");
        f.setSize(400, 300);
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.add(new JLabel("Questa è un'Applicazione scritta con Swing"));
        f.setVisible(true);
    }
}
```

In questo esempio abbiamo le classi **JFrame** e **JLabel**. La classe **JFrame** crea una finestra, mentre la classe **JLabel** crea un'etichetta.

SWT

SWT sta per Standard Widget Toolkit e appartiene alla categoria degli Heavy Weight toolkit.

SWT contiene i migliori componenti di AWT e Swing.

Quindi, quando sviluppiamo un'applicazione SWT, possiamo inserirci al suo interno sia componenti Swing, sia componenti AWT.

TESTING DEL SOFTWARE CON JUNIT

INTRODUZIONE AGLI UNIT TEST

Quando sviluppiamo un software, indipendentemente dal tipo di software che stiamo sviluppando, la fase di test svolge un ruolo fondamentale.

La fase di test consiste nel verificare che il codice che abbiamo scritto ritorna dei risultati attesi.

Già nella fase di sviluppo, una prima parte di test viene fatta attraverso il debug, che verifica se le istruzioni vengano eseguite secondo le nostre indicazioni durante l'esecuzione del software. Tuttavia, la fase di debug è legata alla fase di sviluppo e rimane circoscritta in quella fase. Per eseguire dei test più complessi, è necessario utilizzare degli strumenti appositi, che consentono appunto di validare tutto ciò che abbiamo scritto e certificare che il software che abbiamo scritto esegua esattamente le istruzioni che ci aspettiamo.

Molto spesso la fase di test purtroppo viene molto sottovalutata ed è limitata alla fase di debug, o a pochi test successivi.

Le best practice per lo sviluppo del software prevedono la realizzazione di test su ogni singolo componente (ogni classe e ogni metodo). In questo modo ci assicuriamo che ogni singolo componente presente all'interno del nostro software funzioni come previsto.

Abbiamo due modalità per fare test:

- test manuali, che si effettuano creando una classe con il metodo main() e utilizzando semplicemente tutte le nostre classi, invocando tutti i metodi definiti. Ovviamente, invocando tutti i metodi, possiamo verificare che l'output è quello atteso. Il grosso limite di questo approccio alla realizzazione dei test è che non possiamo ripeterlo nel tempo se il software cambia;
- utilizzo di strumenti ad hoc, che sono software o librerie che supportano gli sviluppatori nella definizione degli unit test. Ad esempio, Junit è un framework per sviluppare unit test in JAVA.

Lo unit test è una metodologia che consente di verificare che una singola unità di codice funzioni correttamente, sulla base di determinate condizioni.

In JAVA, l'unità di codice è un metodo o un gruppo di metodi, presenti in una classe, che implementano una singola funzionalità.

Per esempio, supponiamo di aver implementato una classe che contiene un metodo che verifica se una stringa è lunga almeno 8 caratteri. Creare uno unit test vuol dire creare un test che verifichi che il metodo restituisce false, se passiamo una stringa con meno di 8 caratteri, altrimenti true.

Lo unit test può essere organizzato in:

- Test Case, che consente di verificare una singola unità di codice, quindi un metodo per esempio;
- Test Suite, che è un gruppo di Test Case che consente di verificare diverse funzionalità correlate tra loro. Quindi, consiste semplicemente nell'eseguire più Test Case simultaneamente e correlati tra loro. Immaginiamo il caso in cui abbiamo delle classi che hanno dei metodi che vengono chiamati in cascata. Attraverso il Test Case possiamo creare i singoli unit test dei vari metodi, mentre il Test Suite consente di eseguire tutti questi Test Case simultaneamente.

Vediamo quali sono i vantaggi nell'utilizzo degli unit test:

- rende più semplici le modifiche, perché comunque è possibile rieseguire quegli unit test;
- rende più semplice l'integrazione tra i componenti, perché grazie agli unit test limitiamo i bug legati all'interazione tra i vari componenti;
- è un supporto alla documentazione del software. Essendo questo unit test un esempio di utilizzo della funzionalità, è un prezioso strumento di supporto alla stesura della documentazione.

Vediamo quali sono i limiti dello unit test:

- non sono sempre in grado di identificare tutti gli errori, poiché uno unit test analizza il singolo metodo. Analizzando il singolo metodo, non identifica gli errori legati all'integrazione tra componenti, ma è solo di supporto grazie ai Test Suite;
- non valuta le performance di una funzionalità.

INTRODUZIONE AL FRAMEWORK JUNIT

Junit è un framework open-source che consente di scrivere unit test per software scritto in JAVA.

Junit consente di scrivere ed eseguire Test Case e Test Suite su metodi di classi JAVA.

Junit è arrivato alla versione 5, ma è ancora in beta.

Per ogni classe che abbiamo creato, dobbiamo creare una classe di tipo Test Case, al cui interno possiamo inserire tutti i test da effettuare. Se abbiamo più Test Case correlati, dobbiamo creare una classe Test Suite che conterrà il riferimento alle classi Test Case.

Generalmente, le classi di test vanno inserite in un'apposita cartella, che chiameremo test.

Il nome della classe Test Case generalmente appartiene allo stesso package della classe che vogliamo testare ed ha lo stesso nome della classe che stiamo testando, seguita dalla parola Test. Vediamo un esempio:

- Classe da testare: Persona e si trova nel package it.corso.java nella direcotry src.
- Classe Test Case: PersonaTest si troverà nel package it.corso.java nella directory test.

Le annotation disponibili in Junit 4 sono:

- @Test: indica che il metodo è un test;
- @Before: indica che il metodo deve essere eseguito prima di tutti test;
- @After: indica che il metodo deve essere eseguito dopo tutti i test;
- @BeforeClass: indica che il metodo deve essere eseguito prima di eseguire il primo test;
- @AfterClass: indica che il metodo deve essere eseguito dopo l'esecuzione dell'ultimo test.

All'interno della classe TestCase, dobbiamo invocare il metodo della classe da testare, e dobbiamo verificare che il risultato che ci ritorna il metodo è quello che ci aspettiamo. Se il confronto ha esito positivo, il test ha successo, altrimenti il test fallisce. Tale confronto avviene attraverso dei metodi statici messi a

disposizione da una classe particolare, ovvero la classe `Assert`. Questi metodi ci consentono di verificare che l'output ritornato da un metodo testato corrisponde all'output atteso.

Vediamo un esempio di Test Case :

Esempio

```
package it.preite;
public class Calcolatrice {
    public static void main(String[] args) {
        Calcolatrice c = new Calcolatrice();
        c.somma(2, 3);
    }
    public double somma(double a, double b) {
        return a+b;
    }
}
```

Classe che rappresenta una calcolatrice.

Il metodo `somma` effettua la somma tra due numeri `double`.

```
package it.preite;
import org.junit.Test;
import static org.junit.Assert.*;
public class CalcolatriceTest {
    @Test
    public void verificaSomma() {
        Calcolatrice c = new Calcolatrice();
        double a = 10;
        double b = 5;
        assertEquals("somma corretta", a+b, c.somma(a, b), 0);
    }
}
```

Classe che rappresenta il Test Case della classe `Calcolatrice`.

Il test `verificaSomma` utilizza il metodo statico `assertEquals` per verificare che il metodo `somma` della classe `Calcolatrice` restituisca effettivamente la somma...

Qui abbiamo la classe `Calcolatrice` con un metodo `somma()`, che ritorna semplicemente $a + b$. Dopodichè abbiamo anche la classe `CalcolatriceTest` che ha un metodo `verificaSomma()`. Per quanto riguarda i metodi dei test, possiamo denominarli come vogliamo. Il metodo `verificaSomma()` crea un oggetto di tipo `Calcolatrice` e usa il metodo `assertEquals()` della classe `Assert`, per verificare che la somma $a+b$ (le due variabili di `verificaSomma()`) sia uguale al valore ritornato dalla chiamata del metodo `somma()` della classe `Calcolatrice`.

L'esecuzione di un test genera un risultato che può essere:

- **Successo** (colore verde): il risultato atteso coincide con il risultato ottenuto dal metodo testato.
- **Fallito** (colore rosso): il risultato atteso non coincide con il risultato ottenuto dal metodo testato.
- **Errore** (colore blu): il test è andato in errore. Le cause possono essere diverse:
 - è stata generata un'eccezione durante l'esecuzione del test;
 - il test non è stato configurato correttamente.

Vediamo un altro esempio di Test Case:

Stringa.java

```
public class Stringa {  
  
    public static void main(String[] args) {  
        Stringa stringa = new Stringa();  
  
        int oc = stringa.calcoloNumeroOccorrenze("Lorem ipsum test prova Paolo", "m");  
  
        System.out.println(oc);  
    }  
  
    public int calcoloNumeroOccorrenze(String str, String token){  
        int nOccorrenze = 0;  
  
        for (int i = 0; i <= str.length()-token.length(); i++){  
            String temp = str.substring(i, i + token.length());  
            if(temp.equals(token)){  
                nOccorrenze++;  
            }  
        }  
        return nOccorrenze;  
    }  
}
```

StringaTest.java

```
import org.junit.jupiter.api.Test;  
import variabile.Stringhe;  
  
import static org.junit.jupiter.api.Assertions.*;  
  
class StringaTest{  
  
    @Test  
    public void verificaNumeroOccorrenze(){  
        Stringa stringa = new Stringa();  
  
        String testo = "Oggi è una giornata di sole";  
  
        int occorrenzeCheCiSiAspetta = 3;  
        assertEquals(occorrenzeCheCiSiAspetta,  
stringa.calcoloNumeroOccorrenze(testo, "g"));  
  
    }  
}
```

La Test Suite è una classe che consente di eseguire un insieme di Test Case. Attraverso la classe Test Suite, possiamo eseguire tutti i test associati ad essa, senza necessità di eseguirli uno per volta.

Per naming convention è previsto che la classe Test Suite abbia un nome qualunque, purché termini con Tests.

Esempio

Test Case 1

```
public class CalcolatriceTest {  
    @Test  
    public void verificaSomma() {  
        ...  
    }  
}
```

Test Case 2

```
public class GeometriaTest {  
    @Test  
    public void verificaCalcolaAreaRettangolo () {  
        ...  
    }  
}
```

Test Suite

```
import org.junit.runner.RunWith;  
import org.junit.runners.Suite;  
import org.junit.runners.Suite.SuiteClasses;  
  
@RunWith(Suite.class)  
@SuiteClasses(  
    { CalcolatriceTest.class, GeometriaTest.class })  
public class EseguiTests {  
}
```

Lanciando la classe EseguiTests, verranno eseguiti i Test Case CalcolatriceTest e GeometriaTest

CLASSE ASSERT

La classe Assert, che si trova anch'essa nel package org.junit, contiene una serie di metodi statici che consentono di verificare che la classe che stiamo testando funzioni correttamente.

Questi metodi consentono di confrontare il risultato restituito dal metodo che stiamo testando con l'output atteso.

Se l'assert è vero, il test case continua l'esecuzione.

Se l'assert è falso, il test case viene interrotto in quel punto e il risultato sarà "Fallito".

Se nessun assert fallisce durante l'esecuzione del test case, il risultato sarà "Successo".

Per assert si intende ogni qualvolta viene richiamato un metodo della classe Assert. Se un assert fallisce, tutti quelli presenti dopo non vengono neanche eseguiti.

Vediamo alcuni metodi della classe Assert:

- fail(String msg), che prende in ingresso un messaggio. Questo metodo fa fallire il metodo, indipendentemente dall'esito di tutti gli altri. La variabile msg contiene il testo del messaggio da visualizzare quando il test fallisce;

- `assertTrue(String msg, boolean cond)`, consente di verificare se la condizione cond è vera. La variabile msg contiene il testo del messaggio da visualizzare quando il test fallisce;
- `assertFalse(String msg, boolean cond)`, è l'esatto contrario di `assertTrue()`;
- `assertNull(String msg, Object obj)`, consente di verificare se l'oggetto obj è nullo. Se l'oggetto non è null, il test fallisce.
- `assertNotNull(String msg, Object obj)`, è l'esatto contrario di `assertNull()`.
- `assertEquals(String msg, Object expected, Object actual)`, consente di verificare se expected è uguale ad actual, ovvero se `actual.equals(expected)` ritorna true.
- `assertNotEquals(String msg, Object expected, Object actual)`, è l'esatto contrario di `assertEquals()`;
- `assertSame(String msg, Object expected, Object actual)`, consente di verificare se `actual == expected`;
- `assertArrayEquals(String msg, Object[] expecteds, Object[] actuals)`, consente di verificare se i due array sono uguali, ovvero se:
 - gli array hanno la stessa lunghezza;
 - per ogni elemento i-esimo, viene verificata una delle seguenti condizioni:
 - `assertEquals(expected[i], actual[i])`
 - `assertArrayEquals(expected[i], actual[i]);`

Attraverso Junit è possibile verificare anche che i nostri metodi lancino le eccezioni attese. Vediamo un esempio:

Classe da testare

```
public class Calcolatrice {
    public double divisione(double a, double b)
        throws DivisioneException {
        if(b == 0) {
            throw new DivisioneException();
        }
        return a/b;
    }
}
```

Test Case

```
public class CalcolatriceTest {
    @Test
    public void verificaDivisione() throws DivisioneException {
        Calcolatrice c = new Calcolatrice();
        a = 10;
        b = 0;
        assertEquals(
            "divisione corretta", a/b, c.divisione(a, b), 0);
    }
}
```

Lanciando il Test Case con b=0, il test fallirà a causa del lancio dell'eccezione
DivisioneException



UNIT TEST PARAMETRIZZATO

In JUnit4 è possibile creare test parametrizzati. I test parametrizzati sono classi che consentono di effettuare lo stesso test più volte, passando ogni volta un valore diverso.

Per creare un test parametrizzato è necessario:

- annotare la classe con l'annotation `@RunWith(Parameterized.class)`. Per usare questa annotation dobbiamo importare:
 - `org.junit.runner.RunWith`
 - `org.junit.runners.Parameterized`

```
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;

@RunWith(Parameterized.class)
public class MathUtilTest {
    ...
}
```

- Dopo aver creato la classe con questa annotation, dobbiamo creare un metodo statico che ritorni una Collection. Questo metodo dovrà contenere tutti i valori che passeremo al metodo di test e dovrà essere annotato con `@Parameterized.Parameters`. L'array da passare è un array bidimensionale in cui, per ogni singola riga, il primo elemento è il valore passato e il secondo elemento è l'esito.

```
...
@RunWith(Parameterized.class)
public class MathUtilTest {
    @Parameterized.Parameters
    public static Collection dataset() {
        return Arrays.asList(new Object[][] {
            {1, false},
            {2, true},
            ...
            {10, true}
        });
    }
}
```

- Dopo aver creato questo metodo, dobbiamo creare il costruttore della classe Test Case con due parametri in ingresso, ossia il valore testato e il risultato atteso. Questi valori dovranno poi essere passati a due variabili di istanza che abbiamo precedentemente creato.

```
...
@RunWith(Parameterized.class)
public class MathUtilTest {
    private int numeroTestato;
    private boolean risultatoAtteso;

    public MathUtilTest(int numeroTestato, boolean risultatoAtteso) {
        super();
        this.numeroTestato = numeroTestato;
        this.risultatoAtteso = risultatoAtteso;
    }
}
```

- A questo punto è possibile creare il nostro test nel modo che già sappiamo, passando le due variabili definite in precedenza. Una volta che questo test viene lanciato, viene eseguito tante volte per quanti elementi sono presenti nella Collection che abbiamo configurato.

```
...
@RunWith(Parameterized.class)
public class MathUtilTest {
    @Test
    public void testNumeroPari() {
        System.out.println("... nel test è : " + numeroTestato);

        assertEquals(
            "Il numero " + numeroTestato + " è DISPARI!",
            risultatoAtteso,
            mathUtil.numeroPari(numeroTestato));
    }
}
```

Lanciando la classe, verrà eseguito il test tante volte per quanti sono gli elementi presenti nella collection che rappresenta il nostro dataset!

Vediamo un esempio di unit test parametrizzato:

NumeriPari.java

```
public class NumeriPari {  
  
    public boolean numeroPari(int num){  
        if (num % 2 == 0){  
            return true;  
        }  
        return false;  
    }  
}
```

NumeriPariTest.java

```
import static org.junit.Assert.*;  
  
import org.junit.Test;  
import org.junit.runner.RunWith;  
import org.junit.runners.Parameterized;  
  
import java.util.Arrays;  
import java.util.Collection;  
  
@RunWith(Parameterized.class)  
public class NumeriPariTest{  
    private int numeroTestato;  
    private boolean risultatoAtteso;  
  
    @Parameterized.Parameters  
    public static Collection<Object[]> getParam() {  
        return Arrays.asList(new Object[][] {  
            {1, false},  
            {2, true},  
            {3, false},  
            {4, true},  
            {5, false},  
            {6, true},  
            {7, false},  
            {8, true},  
            {9, false},  
            {10, true}  
        });  
    }  
  
    public NumeriPariTest(int numeroTestato, boolean risultatoAtteso) {  
        this.numeroTestato = numeroTestato;  
        this.risultatoAtteso = risultatoAtteso;  
    }  
  
    @Test  
    public void testNumeroPari() {  
        NumeriPari numeriPari = new NumeriPari();  
        System.out.println("Eseguo il test con il numero " + numeroTestato);  
        assertEquals(risultatoAtteso, numeriPari.numeroPari(numeroTestato));  
    }  
}
```

STORIA DEL WEB

Oggi usiamo il web per vari scopi:

- lavoro;
- video;
- musica;
- social network;
- divertimento;
- ...

Tutto questo è possibile grazie a Internet.

Per capire come siamo arrivati a questa diffusione, dobbiamo fare qualche passo indietro di qualche anno.

Il World Wide Web (WWW) nasce ufficialmente il 6 Agosto del 1991, quando Tim Berners-Lee (un fondatore del World Wide Web) pubblicò il primo sito web denominato info.cern.ch . Questo sito web è ancora tuttora navigabile.

L'idea del web nacque nel 1989 presso il CERN di Ginevra. In quel contesto, Tim Berners-Lee notò che alcuni colleghi italiani trasmettevano informazioni a video da un piano all'altro attraverso una linea telefonica.

A partire da quella situazione, Tim Berners-Lee presentò un progetto che consisteva nell'elaborare un software in grado di condividere la documentazione scientifica tra tutti i ricercatori. Ovviamente, questo software doveva essere indipendente dalla piattaforma utilizzata.

L'obiettivo iniziale del progetto era quello di migliorare la comunicazione tra i ricercatori.

Parallelamente alla creazione del software, ebbe inizio anche la definizione di standard e protocolli per lo scambio di documenti attraverso una rete di calcolatori. Una rete di calcolatori è una rete in cui ci sono più computer connessi tra di loro.

Questi standard su cui si appoggia tutto il mondo del web sono il linguaggio HTML e il protocollo di rete HTTP.

Il software pensato per condividere la documentazione scientifica era sostanzialmente un sito web.

Inizialmente i protocolli erano in grado di gestire solo pagine HTML statiche, ovvero dei file che venivano copiati all'interno di un server e accessibili mediante un browser web.

L'evoluzione del web ha poi portato alla creazione di pagine dinamiche, attività di interazione con l'utente, pagine asincrone e così via.

Il 30 Aprile del 1993 il CERN mise il WWW a disposizione del pubblico. Da quel giorno è iniziata quella che ora conosciamo come "era del web".

APPLICAZIONI CLIENT / SERVER

Prima del web, le applicazioni in rete erano gestite attraverso un modello architettonico chiamato client / server. Ancora oggi ci sono delle applicazioni che utilizzano questo approccio.

L'architettura client / server è composta da un componente centrale, chiamato server, e da uno o più componenti locali, chiamati client o terminale. Su ogni componente di questa architettura deve essere installato un software.

Se pensiamo un attimo al web, il principio non è molto diverso, perché abbiamo un server su cui sono installate tutte le web application e i siti web, e tanti client (i browser) che sono in grado di elaborare le richieste.

I limiti dell'architettura client / server sono:

- la necessità di installare un software su ciascun client;
- la necessità di sviluppare un software per i client per ogni sistema operativo.

STANDARD WEB

Gli standard più importanti utilizzati nel web sono:

- HTML: è linguaggio XML based, cioè formato da tanti elementi chiamati tag, con cui vengono scritte le pagine web;
- HTTP: è il protocollo di rete su cui si basa il web;
- URL (Uniform Resource Locator): è una stringa di caratteri che identifica univocamente una risorsa (pagina web, video, immagine, ...) all'interno della rete internet.

PROTOCOLLI DI RETE

Un protocollo è un'insieme di regole che definiscono uno standard di comunicazione tra diversi computer attraverso la rete.

Banalmente, quando parliamo con un'altra persona in italiano, utilizziamo le regole di grammatica che rappresentano lo standard per utilizzare tale linguaggio. Questo è lo stesso principio che vale per il protocollo.

Quindi, il protocollo definisce le regole che devono rispettare due o più computer, che si scambiano messaggi tra di loro.

Tutta la rete internet è regolata dalla famiglia di protocolli TCP/IP.

Un protocollo descrive:

- il formato del messaggio;
- il modo con cui devono essere scambiati i messaggi tra i vari computer nella rete.

Per ogni servizio che viene erogato tramite internet, esiste un protocollo, ovvero esistono delle regole di scambio dei messaggi. Per servizio intendiamo inviare email, stabilire connessioni remote, trasferire file, fare l'upload di file e così via.

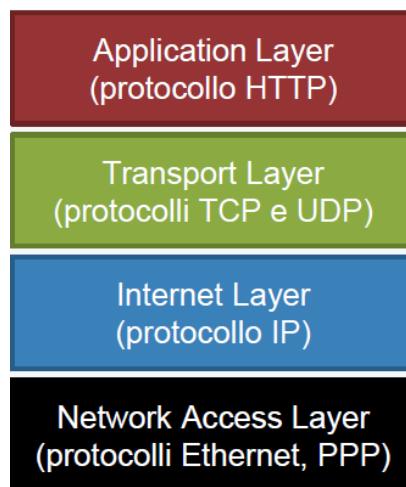
Alcuni protocolli utilizzati sono:

- Simple Mail Transfer Protocol (SMTP), per la gestione dei messaggi di posta elettronica;
- File Transfer Protocol (FTP), per il trasferimento di file tra le macchine remote;
- Hypertext Transfer Protocol (HTTP), per la trasmissione di informazioni attraverso il web.

PILA PROTOCOLLARE TCP / IP

Abbiamo detto che tutto internet si basa su un set di protocolli appartenenti alla famiglia TCP / IP.

Il TCP / IP è, appunto, un insieme di protocolli utilizzati per lo scambio di dati in rete, ed è formato da 4 livelli principali:



- Application Layer: è il livello più alto, cioè quello che viene interpretato dal browser, gestito attraverso il protocollo HTTP;
- Transport Layer: è il livello un po' più basso rispetto l'Application Layer ed è gestito attraverso i protocolli TCP e UDP;
- Internet Layer: è gestito attraverso il protocollo IP;
- Network Access Layer: è il livello più basso ed è gestito attraverso i protocolli Ethernet e PPP.

I protocolli più importanti sono:

- TCP e UDP che organizzano la suddivisione (frammentazione) in pacchetti dei dati da inviare;
- IP che gestisce l'instradamento dei pacchetti dal server al client e viceversa.

PROTOCOLLO TCP

TCP (Transmission Control Protocol) è il protocollo usato comunemente su internet.

Quando viene richiesta una risorsa al server web, il client invia dei pacchetti TCP, richiedendo la risorsa.

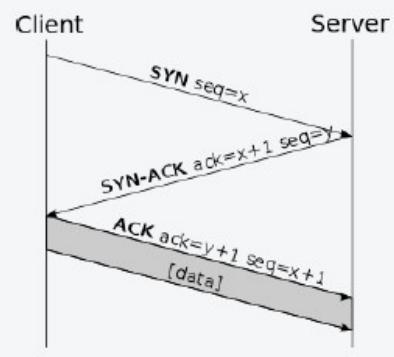
Il web server risponde inviando a sua volta un flusso di altri pacchetti TCP.

Il client prende i pacchetti ricevuti e visualizza la risorsa dell'utente (pagina web, immagine, file scaricato,...).

Il protocollo TCP garantisce che il client riceva tutti i pacchetti, senza perdita o danneggiamento.

Il protocollo TCP utilizza uno schema detto Three Way Handshake.

- Il Client invia un segmento SYN al Server
- Il Server invia un segmento SYN/ACK al Client
- Il Client invia un segmento ACK al Server
- A questo punto la connessione è stabilita



Abbiamo una fase iniziale in cui client e server si scambiano dei messaggi. Questa fase è il principio di funzionamento del ping, tramite il quale contatto il server, quest'ultimo mi risponde e io rispondo ulteriormente.

Terminata questa fase di Handshake a tre vie, inizia la trasmissione dei dati. Per esempio, se digito un' URL, il mio client contatta prima il server, si scambiano dei messaggi e, solo a questo punto, invia la richiesta contenente la URL e tutte le informazioni che ho inviato al server. Il principio di funzionamento rimane identico anche dal server al client.

PROTOCOLLO UDP

Il protocollo UDP (User Datagram Protocol) ha lo stesso principio di funzionamento del protocollo TCP, con la differenza che l'UDP non controlla gli errori. Quindi, con il protocollo UDP non abbiamo la garanzia che tutti i pacchetti inviati dal server siano stati ricevuti.

Non avendo un controllo sugli errori, il flusso dei dati è molto più rapido rispetto il protocollo TCP.

Il protocollo UDP è utilizzato principalmente per lo streaming, video, audio, giochi online e così via.

PORTE DEL PROTOCOLLO TCP / IP

Le porte del protocollo TCP / IP sono componenti software utilizzati per identificare, tramite dei numeri, la connessione di trasporto.

Tali porte servono per identificare, su un computer, l'applicazione a cui instradare il flusso dei dati. Se non avessimo queste porte, il computer potrebbe gestire solo un'applicazione per volta. Attraverso le porte, quindi, siamo in grado di gestire il flusso dati che arriva da internet e di reindirizzarlo correttamente all'applicazione che ha fatto la richiesta di accesso alle informazioni attraverso il canale internet.

L' Internet Assigned Numbers Authority (IANA) è un ente che si occupa di standardizzare le porte e di aggiornare un documento che contiene l'elenco di queste porte. Questo documento si chiama ports-number.

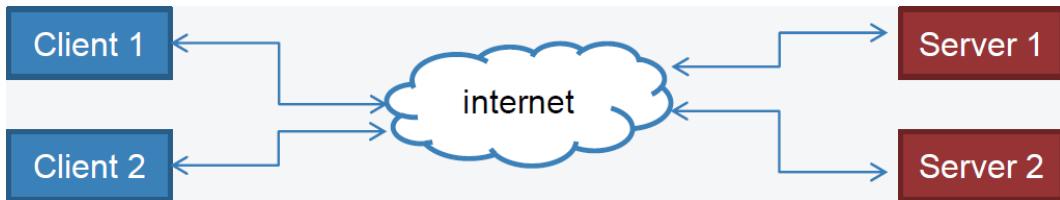
Esempi di porte e relative protocolli

21	FTP (File Transfer Protocol)
22	SSH (Secure Shell)
25	SMTP (Simple Mail Transfer Protocol)
80	HTTP (Hyper Text Transfer Protocol)
110	POP3 (Post Office Protocol 3)
143	IMAP (Internet Message Access Protocol)
443	HTTPS (Secure HTTP)

PROTOCOLLO HTTP

Il protocollo HTTP (HyperText Transfer Protocol) è un protocollo utilizzato per la trasmissione di informazioni sul web.

Il protocollo HTTP è stato creato per gestire il trasferimento di informazioni in formato HTML.



Quindi, un client (browser) e i server web, che devono erogare contenuti all'interno di internet, devono essere in grado di gestire il protocollo HTTP, per cui devono essere in grado di scambiarsi dei messaggi contenenti documenti in formato HTML.

Il protocollo HTTP è un protocollo “stateless”, cioè senza memoria. Capire che il protocollo HTTP è stateless è una cosa fondamentale. In particolare, stateless vuol dire il protocollo non conserva memoria tra una connessione e l'altra, cioè tra una richiesta e l'altra.



Infatti, quando il browser effettua la richiesta, il server elabora la richiesta, restituisce al browser un documento in formato HTML e chiude la connessione. Dalla seconda richiesta in poi, il server non è in grado di capire chi ha effettuato la richiesta e deve nuovamente effettuare una connessione, per poi chiuderla. Quindi, ad ogni richiesta, il server deve aprire la connessione e chiuderla.

Questa caratteristica stateless del protocollo HTTP, insieme al Three Way Handshake del protocollo TCP, aumenta i tempi di risposta del server.

Il protocollo HTTP, appunto, si basa sul meccanismo di richiesta/risposta tra client e server, dove:

- il client (tipicamente un browser) esegue una richiesta;
- il server (il web server) riceve la richiesta del client e restituisce una risposta.

Quindi, possiamo distinguere due tipi di messaggio:

- messaggi di richiesta (o HTTP request), inviato dal client al server;
- messaggi di risposta (o HTTP response), inviato dal server al client.

Il messaggio di richiesta (HTTP request) è composto da tre parti:

- request line: che contiene il metodo (GET, POST e così via), la risorsa richiesta al server chiamata URI (uniform resource identifier) e la versione del protocollo HTTP;
- header: che contiene una serie di informazioni, tra cui l'User-Agent (browser) e l'Host (server a cui vogliamo collegarci);
- body: che contiene il corpo del messaggio, quindi le informazioni che stiamo richiedendo al server.

Il metodo GET consente di ottenere il contenuto della risorsa associata all'URI.

Il metodo POST è usato generalmente per inviare informazioni al server, tipicamente i dati di un form.

Il messaggio di risposta (HTTP response) è composto da tre parti:

- status-line (o status-code): che contiene un codice a tre cifre che indica l'esito della ricezione della richiesta, quindi se è andata a buon fine o se ci sono stati degli errori;
- header: che contiene una serie di informazioni, tra cui il tipo o la versione del server e il tipo del contenuto denominato content-type (pdf, immagini e così via);
- body: che contiene il contenuto della risposta.

Gli **status-code più comuni** sono:

- **200 OK.** La richiesta è stata ricevuta, capita e accettata.
- **301 Moved Permanently.** La risorsa richiesta non è raggiungibile perché è stata spostata in modo permanente.
- **400 Bad Request.** La risorsa richiesta non è comprensibile al server.
- **404 Not Found.** La risorsa richiesta non è stata trovata dal server.
- **500 Internal Server Error.** Si è verificato un problema interno ed il server non è in grado di rispondere alla richiesta (generalmente si ha quando si verifica un errore applicativo).

WEB APPLICATION VS SITO STATICO

Un'applicazione web è un software web-oriented, in grado di generare contenuti web dinamici e scambiare informazioni con l'utente. Alcuni esempi di applicazioni web sono i sistemi di pagamento, i Google Docs e così via.

Un'applicazione web è accessibile generalmente tramite internet, ma ci sono dei casi in cui tali applicazioni sono accessibili localmente (senza internet), installando un server in una rete locale.

Le prime applicazioni web generavano pagine HTML, mentre oggi sono in grado di scambiare altri tipi di messaggi, come JSON o XML.

JSON e XML sono i formati dei dati messi a disposizione dalle applicazioni web.

Oggi si parla molto di applicazioni a microservizi, dove un microservizio è un componente di un'applicazione in grado di erogare contenuti in formato JSON, o XML.

Il sito statico è composto da pagine HTML già pronte e copiate all'interno di una cartella del nostro web server.

Il sito statico non è in grado di elaborare dati inseriti dall'utente. Quindi, se abbiamo un form e inviamo i dati, il sito statico non è in grado di ricevere questi dati ed elaborarli.

L'unica interazione che è concessa con un sito statico è la consultazione, quindi vedere un contenuto HTML e navigare tra varie pagine. Ogni aggiornamento sul sito statico deve essere effettuato manualmente nel file HTML.

Nell'applicazione web, invece, le pagine sono generate dinamicamente. Le pagine di un'applicazione web sono composte da porzioni di codice HTML, insieme a porzioni di codice di linguaggi di programmazione (JAVA, PHP e così via) che consentono di generare altro output sulla base delle informazioni inviate dall'utente, oppure sulla base di informazioni elaborate dal server.

Infine, un'applicazione web è in grado di interagire con l'utente, riuscendo per esempio a ricevere ed elaborare i dati che gli sono stati inviati da un form.

Vantaggi Sito statico

- Velocità di visualizzazione delle pagine

Svantaggi Sito statico

- Difficoltà di manutenzione
- Impossibile creare applicazioni complesse

Vantaggi Applicazione web

- Manutenzione più facile
- Possibilità di creare applicazioni complesse

Svantaggi Applicazione web

- Velocità di visualizzazione delle pagine

ELABORAZIONE DINAMICA DELLA RISPOSTA

CLIENT WEB

Il client web, in particolare il web browser, è un software che consente di accedere ad applicazioni web, a siti statici ed altre risorse erogate dai server (immagini, video, ...).

Il browser svolge due compiti principali:

- è la componente client del protocollo HTTP e, in quanto tale, gestisce il download delle risorse dal server web;
- fa visualizzare all'utente le risorse scaricate, quindi fa visualizzare i contenuti HTML e riproduce quelli multimediali.



SERVER WEB

Il server web è un software all'interno del quale sono installate le applicazioni web.

Il server web gestisce le richieste di pagine web effettuate da un client.

La comunicazione tra client e server, abbiamo già detto, avviene tramite il protocollo HTTP.



Il protocollo HTTP utilizza la porta 80 per trasmettere i dati al web server. Alcune volte viene utilizzata la porta 8080, soprattutto nei casi in cui si utilizza il linguaggio JAVA.

In caso di utilizzo del protocollo HTTP sicuro (HTTPS), viene utilizzata la porta 443.

Tutti i web server interconnessi a livello mondiale costituiscono il World Wide Web.

DALLA RICHIESTA ALLA RISPOSTA

L'accesso ad una risorsa web può avvenire:

- digitando l'URL nel browser web;
- cliccando su un collegamento ipertestuale (link) presente in una pagina già visualizzata;
- cliccando su un collegamento ipertestuale presente in altre risorse (ad esempio una email).

Quando richiediamo l'accesso ad una risorsa (ad un'URL), il browser inizia il processo di richiesta della risorsa al server.

Browser e server si scambiano una serie di messaggi che si concludono con il download della risorsa sul computer dell'utente.



Di seguito un esempio:

- Definiamo questi elementi per l'esempio :

URL: <http://www.paolopreite.it/index.php/chi-sono>

Server name: www.paolopreite.it

Risorsa richiesta: /index.php/chi-sono

Innanzitutto, il server name viene risolto in un indirizzo IP (ad esempio 89.234.123.33).

L'indirizzo IP è un numero associato ad una scheda di rete fisica.

I server hanno un indirizzo IP pubblico fisso, quindi non può cambiare dinamicamente, altrimenti non riusciremmo a capire mai qual è il server a cui vogliamo indirizzare la richiesta.

Il server name viene convertito in indirizzo IP attraverso il DNS (Domain Name System), cioè un database distribuito a livello mondiale che contiene l'elenco di tutti gli indirizzi IP pubblici associati ai server name.

A questo punto, il browser effettua la richiesta richiedendo il testo HTML e, una volta ricevuto, viene interpretato dal browser. Il browser effettua tante richieste per quante risorse sono presenti nel codice HTML (immagini, video, fogli di stile,...).

Ricevuti tutti i file, il browser visualizza la pagina sullo schermo, sulla base delle specifiche dei linguaggi web (HTML, CSS, ...), e la pagina visualizzata conterrà tutte le risorse contenute nel codice HTML.

LINGUAGGI PER L'ELABORAZIONE DI WEB APPLICATION

Innanzitutto dobbiamo fare una distinzione importante tra linguaggi client side e linguaggi server side.

I linguaggi server side sono linguaggi di programmazione che consentono di creare applicazioni web, che interagiscono in maniera dinamica con il client.

Il linguaggio server side viene interpretato a livello server.

Immaginando con JAVA, per esempio, abbiamo un'application server (o servlet container, come Tomcat) che è in grado di elaborare una richiesta e, attraverso il codice JAVA, di generare un output HTML.

L'elaborazione del codice porta alla generazione di un flusso di codice HTML, o di altro codice (ad esempio JSON o XML) che viene restituito al client.

Esempi di linguaggi server side:

- CGI (Common gateway Interface) e/o Perl: è stata la prima tecnologia in grado di processare le richieste provenienti dal browser restituendo codice HTML;
- Java: offre diversi strumenti e framework per la creazione di web application (Servlet, JSP, Struts, Spring, JSF...);
- PHP: è un linguaggio di scripting interpretato largamente diffuso. I principali CMS di uso comune sono scritti in PHP (Wordpress, Joomla, ...);
- Python: è un linguaggio di programmazione object oriented che tra le altre cose, consente di scrivere web application;
- .NET (di proprietà Microsoft): è il framework creato da Microsoft per la realizzazione di web application.

I linguaggi client side, invece, sono linguaggi di programmazione a livello di browser, quindi consentono di effettuare istruzioni direttamente all'interno del client (all'interno del browser), come la creazione diretta di form, aggiunta o rimozione dinamica di pezzi di pagina e così via.

Esempi di linguaggi client side:

- JavaScript: attualmente è il principale linguaggio client side;
- Flash (ormai in disuso);
- Silverlight: di proprietà di Microsoft.

Una cosa fondamentale da capire è che non è possibile scrivere una web application utilizzando solo un linguaggio client side.

Se dobbiamo scrivere una web application, essa deve interagire con un server. Dalla parte del server, dobbiamo comunque sviluppare una componente server (in qualsiasi linguaggio di programmazione) in grado di elaborare delle richieste.

Ad esempio, se vogliamo sviluppare un gestionale che si colleghi ad un database per gestire gli ordini, il carrello, il catalogo prodotti e così via, non possiamo assolutamente sviluppare una web application fatta solo con javascript, perché ci deve essere comunque una componente server in grado di elaborare la richiesta.

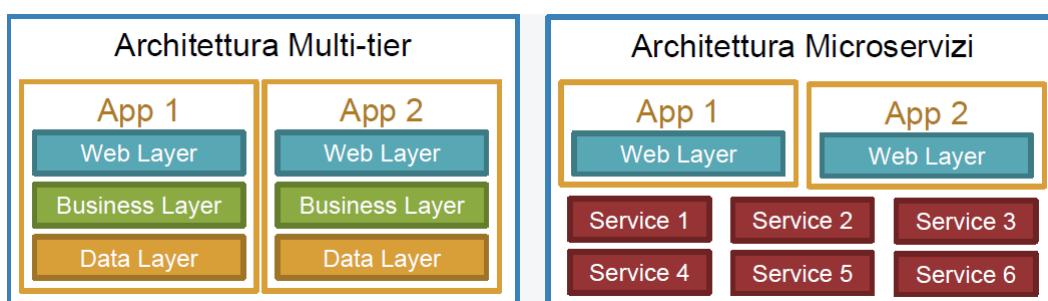
JEE

JEE è l'acronimo di Java Platform Enterprise Edition, ed è un insieme di specifiche tecniche che consentono di sviluppare servizi robusti, sicuri ed efficienti.

Ad oggi JEE è tra le piattaforme tecnologiche più importanti per lo sviluppo di applicazioni in ambito enterprise, cioè applicazioni per banche, assicurazioni, pubblica amministrazione e così via.

Fino a qualche anno fa, le applicazioni che seguivano la specifica JEE erano sviluppate con architettura multi-tier.

L'architettura multi-tier è un'architettura basata su più livelli applicativi, dove ciascun livello prevede il disaccoppiamento tra la parte web (l'interfaccia utente), la parte della logica di business e la parte della gestione dei dati.



Le nuove specifiche JEE si sono evolute nel tempo, consentendo di sviluppare applicazioni basate su architettura a microservizi.

L'architettura a microservizi prevede lo sviluppo di tanti piccoli componenti autonomi in grado di interagire tra loro. Quindi, invece che implementare una parte di logica del business e una parte della gestione dei dati per ogni applicazione che dobbiamo sviluppare, abbiamo l'architettura a microservizi che consente di sviluppare la parte web (con i relativi pattern MVC e così via) all'interno dell'applicazione e una serie di servizi esterni, indipendenti tra loro, che possono essere utilizzati da più applicazioni diverse. Di conseguenza, le parti di logica di business e gestione dei dati vengono staccate dall'applicazione vera e propria.

Questa evoluzione ha portato anche al cambiamento del nome dei software che implementano le specifiche JEE. Prima questi software si chiamavano Application server, ora si chiamano Referencing runtimes.

Le principali specifiche JEE, raggruppate per tipologia sono:

- Specifiche Web Service: consentono di realizzare Web Service REST e SOAP e sono:
 - RESTful Web Services per lo sviluppo di Web Service REST;
 - JAX-WS per lo sviluppo di Web Service SOAP;
 - JSON Processing e JSON Binding per elaborare stringhe JSON e per trasformare oggetti JAVA in stringhe JSON, e viceversa.
 - JAXB per trasformare oggetti JAVA in stringhe XML, o viceversa.
- Specifiche Web: consentono di realizzare componenti relative alla parte di presentation layer (il

presentation layer consente la gestione della sintassi e della semantica delle informazioni trasmesse) e sono:

- Servlet e Java Server Pages;
- JSF (Java Server Faces): framework web basato sul pattern MVC.
- Specifiche Enterprise: consentono di realizzare componenti in contesti applicativi di grandi dimensioni e sono:
 - Contexts and Dependency Injection, Enterprise JavaBeans, Java Message Service, Java EE Security API.
- Specifiche per l'interazione con i database:
 - JTA (Java Transaction API), JDBC(Java DataBase Connectivity), JPA (Java Persistence API).

I principali Referencing Runtimes ad oggi in uso sono:

- GlassFish, che è l'implementazione di riferimento open source mantenuta da Oracle;
- WildFly, che è l'evoluzione di quello che prima si chiamava Jboss ed è anch'esso open source;
- WebLogic, di proprietà di Oracle;
- WebSphere, di proprietà di IBM;
- Apache TomEE che incorpora al suo interno Tomcat e MyFaces. Tomcat implementa le specifiche Servlet e JSP.

APACHE TOMEE

INSTALLAZIONE JDK

Prima di installare Apache TomEE, si deve installare il JDK qualora non ci fosse.

Partiamo quindi con l'andare a scaricare l'eseguibile del JDK, cercando su Google "jdk.java.net" e cliccando sul primo risultato della ricerca. Non utilizzo il sito ufficiale di Oracle, in quanto è richiesto un account per poter scaricare i JDK.

Di seguito la home page del sito cercato:

Production and Early-Access OpenJDK Builds, from Oracle

cliccare su uno dei 2

Ready for use: JDK 19, JDK 18, JMC 8

Early access: **JDK 20, Generational ZGC, JavaFX 20, Jextract, Loom, Metropolis, Panama, & Valhalla**

Looking to learn more about Java? Visit [dev.java](#) for the latest Java developer news and resources.

Looking for Oracle JDK builds and information about Oracle's enterprise Java products and services? Visit the [Oracle JDK Download page](#).

Una volta cliccato dove indicato nell'immagine precedente, arriveremo in un'altra pagina dove è possibile scegliere la versione del jdk che ci serve:

jdk.java.net

OpenJDK JDK 19.0.1 General-Availability Release

This page provides production-ready open-source builds of the Java Development Kit, version 19, an implementation of the Java SE 19 Platform under the GNU General Public License, version 2, with the Classpath Exception.

Commercial builds of JDK 19.0.1 from Oracle, under a non-open-source license, can be found at the [Oracle Technology Network](#).

Documentation

- Features
- Release notes
- API Javadoc

Builds

	Linux/AArch64	tar.gz (sha256)	194660832 bytes
Scegliere versione	Linux/x64	tar.gz (sha256)	195925792
JDK e cliccarci sopra	macOS/AArch64	tar.gz (sha256)	190630653
	macOS/x64	tar.gz (sha256)	192577932
	Windows/x64	zip (sha256)	194441800

Notes

- If you have difficulty downloading any of these files please contact download-help@openjdk.org.

Feedback

If you have suggestions or encounter bugs, please submit them using the usual Java SE bug-reporting channel. Be sure to include complete version information from the output of the `java --version` command.

Scelta la versione del JDK, verremo indirizzati sulla pagina di download, da dove potremo scaricarcelo a seconda della versione del nostro sistema operativo (nel mio caso Windows):

[jdk.java.net](https://jdk.java.net/18/)

Java Platform, Standard Edition 8 Reference Implementations

The official Reference Implementations for Java SE 8 (JSR 337) are based solely upon open-source code available from the JDK 8 Project in the OpenJDK Community. This Reference Implementation applies to JSR 337 Maintenance Release 4 (Jul 2022). Reference Implementation for Maintenance Release 1 (Mar 2015), Maintenance Release 2 (Mar 2019), and Maintenance Release 3 (Feb 2020) contains RIs for these releases. Binaries are provided for both the Linux x64 and Windows i586 platforms and Compact Profiles for Linux i586.

These binaries are for reference use only!

These binaries are provided for use by implementers of the Java SE 8 Platform Specification and are for reference purposes only. These Reference Implementations have been approved through the Java Community Process. Binaries for development and production will be available from Oracle and in most popular Linux distributions.

RI Binaries (build 1.8.0_42-b03) under the GNU General Public License version 2

- Oracle Linux 7.3 x64 Java Development Kit (md5) 167 MB
- Windows 10 i586 Java Development Kit (md5) 92 MB
- Oracle Linux 7.3 i586 Java Runtime Environment for Compact Profiles
 - Compact Profile 1 (md5) 15 MB
 - Compact Profile 2 (md5) 18 MB
 - Compact Profile 3 (md5) 20 MB
 - Full JRE (md5) 40 MB

RI Source Code

The source code of the RI binaries is available under the GPLv2 in a single zip file (md5) 123 MB.

International use restrictions

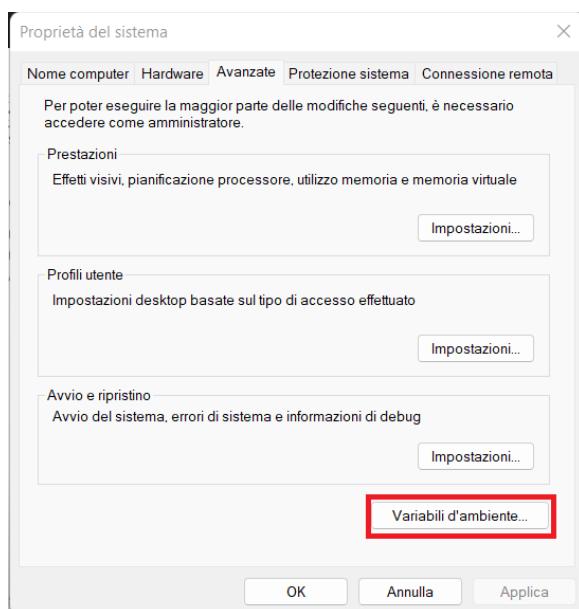
Due to limited intellectual property protection and enforcement in certain countries, the JDK source code may only be distributed to an authorized list of countries. You will not be able to access the source code if you are downloading from a country that is not on this list. We are continuously reviewing this list for addition of other countries.

 © 2022 Oracle Corporation and/or its affiliates
Terms of Use · Privacy · Trademarks

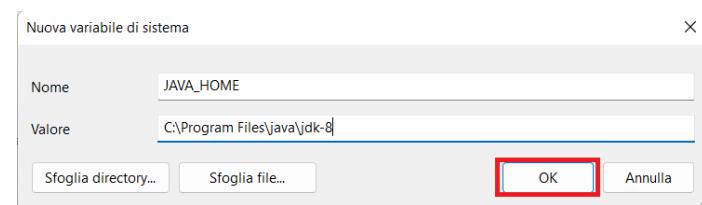
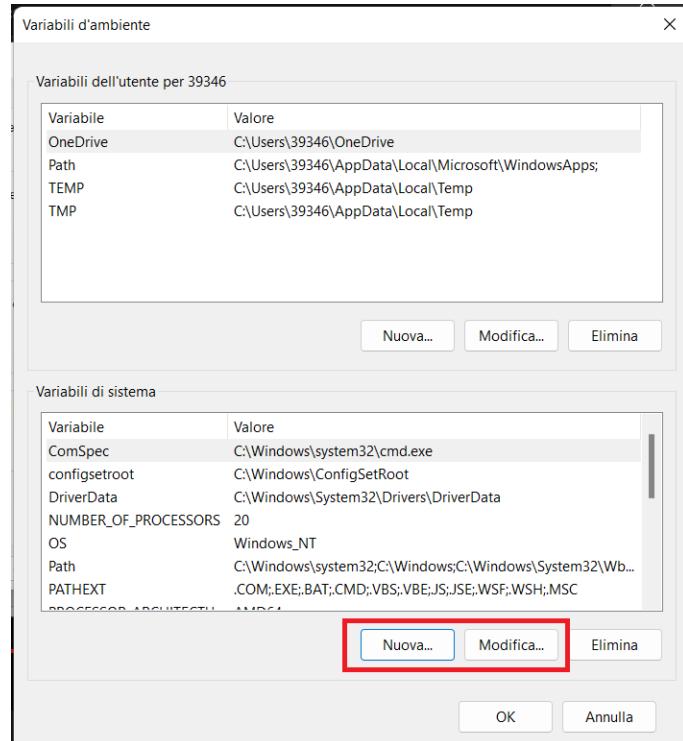
Una volta scaricato, ci ritroveremo un pacchetto .zip che potremo estrarre.

Dopo l'estrazione del pacchetto, dobbiamo spostare la cartella estratta nel percorso “C:\Program Files\java”. Creare la cartella “java” in “Program Files” se ancora non esiste. La cartella “java” potrà contenere anche più versioni del JDK.

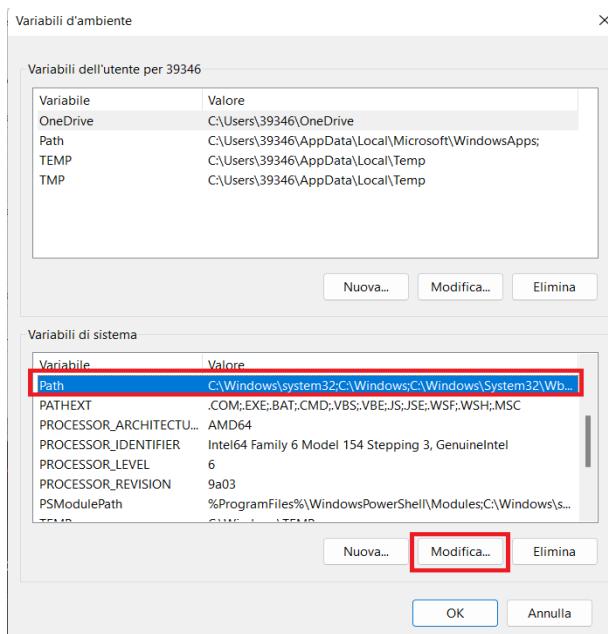
Una volta fatto ciò, va inserito il JDK nelle variabili d'ambiente:



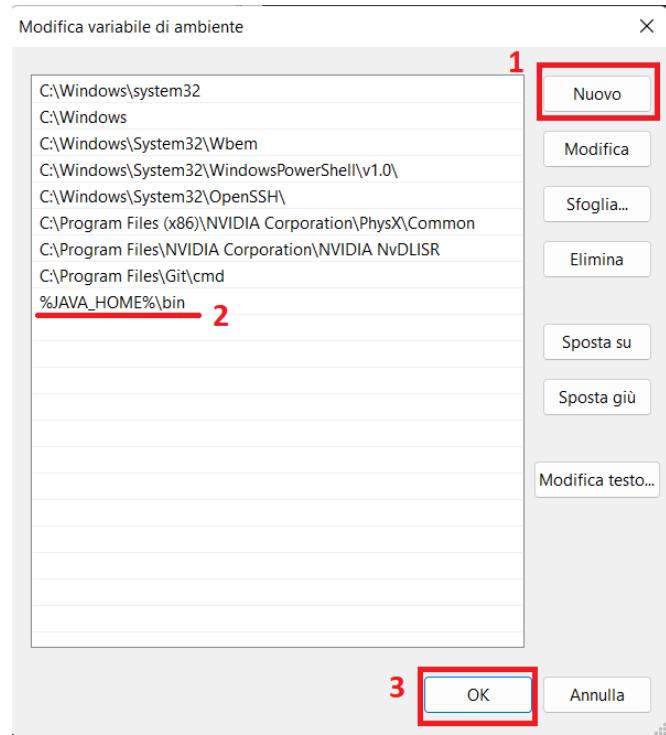
Si deve creare una variabile d'ambiente chiamata "JAVA_HOME", oppure va modificata se già è presente:



Infine, modificare la variabile d'ambiente Path:



e inserirci all'interno la stringa "%JAVA_HOME%\bin" :



Per verificare se l'installazione del JDK ha avuto successo o che la versione installata è quella scelta, utilizzare il comando "java -version" sul prompt:

```
C:\> Prompt dei comandi
Microsoft Windows [Versione 10.0.22000.1098]
(c) Microsoft Corporation. Tutti i diritti riservati.

C:\Users\39346>java -version
openjdk version "1.8.0_42"
OpenJDK Runtime Environment (build 1.8.0_42-b03)
OpenJDK Client VM (build 25.40-b25, mixed mode)
```

Impostare una variabile d'ambiente con la versione scelta del JDK, serve per creare tutte le applicazioni che vogliamo con la versione JAVA scelta di default.

INSTALLAZIONE APACHE TOMEET

Ora che abbiamo installato il JDK e lo abbiamo inserito nella variabili d'ambiente, abbiamo tutto il necessario per poter far funzionare Apache TomEE.

Per scaricare Apache TomEE, dobbiamo andare su tomee.apache.org e cliccare sulla sezione del Download:

The screenshot shows the Apache TomEE homepage with a red background featuring Java code snippets. At the top, there's a navigation bar with links: Documentation, Community, Security, and Downloads (which is highlighted with a red box). Below the navigation, the text "Apache TomEE" and "Now Jakarta EE 9.1 Web Profile Certified!" is prominently displayed. At the bottom, there are three main links: Documentation, Community, and Downloads, each with a small icon and a brief description.

Documentation: Learn more about Apache TomEE

Community: How can I contribute to TomEE?

Downloads: How can I download Apache TomEE?

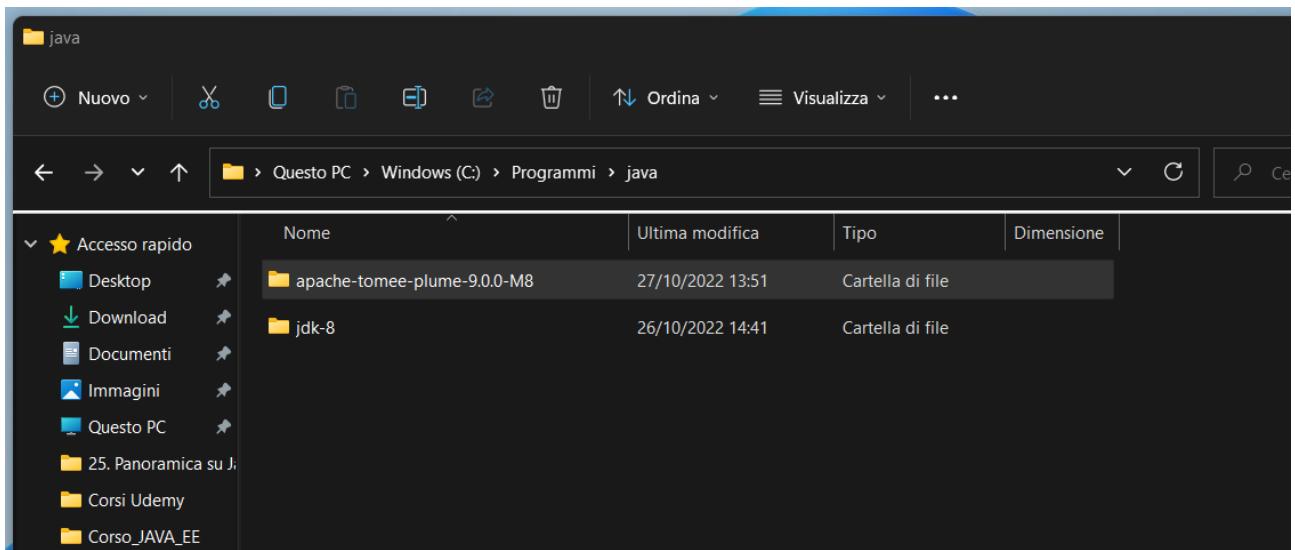
Una volta entrati nella pagina di download, troviamo l'elenco delle varie versioni di TomEE da scaricare. Il consiglio è quello di scaricare la versione TomEE plume, che è quella più ricca di librerie per implementare applicazioni JAVAE:

The screenshot shows the Apache TomEE download page. At the top, there's a navigation bar with links: Documentation, Community, Security, and Downloads (which is highlighted with a red box). Below the navigation, there's a list of download options. The "TomEE Plume ZIP" option is highlighted with a red box. A note at the bottom left indicates that TomEE 9.0.0-M8 is a milestone release targeting the jakarta namespace.

Name	Version	Date	Size	Signatures & Hashes
TomEE Microprofile ZIP	9.0.0-M8	28 Jun 2022	64 MB	PGP SHA512
TomEE Microprofile TAR.GZ	9.0.0-M8	28 Jun 2022	63 MB	PGP SHA512
TomEE Plume ZIP	9.0.0-M8	28 Jun 2022	77 MB	PGP SHA512
TomEE Plume TAR.GZ	9.0.0-M8	28 Jun 2022	76 MB	PGP SHA512
TomEE Plus ZIP	9.0.0-M8	28 Jun 2022	69 MB	PGP SHA512
TomEE Plus TAR.GZ	9.0.0-M8	28 Jun 2022	69 MB	PGP SHA512
TomEE Webprofile ZIP	9.0.0-M8	28 Jun 2022	55 MB	PGP SHA512
TomEE Webprofile TAR.GZ	9.0.0-M8	28 Jun 2022	54 MB	PGP SHA512

! Note: TomEE 9.0.0-M8 is a milestone release, which targets the `jakarta` namespace. Applications using `javax` will not work on this release of TomEE 9. You may encounter issues – feedback on this release is encouraged and appreciated.

Dopo aver scaricato il file zip, lo estraiamo e ne spostiamo il contenuto nella stessa cartella dove abbiamo il JDK:



Questo perché, per far partire le applicazioni JAVAEE con Apache TomEE (o qualsiasi altro Referencing Runtimes), ci serve proprio il JDK, mentre non ci serve il JRE.

Arrivati a questo punto, rimane da configurare Apache TomEE all'interno dell'IDE che si vuole utilizzare.

COMPONENTI DI APACHE TOMEET

Se apriamo la cartella di Apache TomEE, possiamo vedere che ha i seguenti file e cartelle al suo interno:

	Nome	Ultima modifica	Tipo	Dimensione
📁	bin	27/10/2022 13:51	Cartella di file	
📁	conf	27/10/2022 13:51	Cartella di file	
📁	lib	27/10/2022 13:51	Cartella di file	
📁	logs	28/06/2022 10:05	Cartella di file	
📁	temp	27/10/2022 13:51	Cartella di file	
📁	webapps	27/10/2022 13:51	Cartella di file	
📁	work	28/06/2022 10:05	Cartella di file	
📄	BUILDING	27/10/2022 13:51	Documento di testo	20 KB
📄	CONTRIBUTING.md	27/10/2022 13:51	File MD	7 KB
📄	LICENSE	27/10/2022 13:51	File	96 KB
📄	NOTICE	27/10/2022 13:51	File	9 KB
📄	README.md	27/10/2022 13:51	File MD	4 KB
📄	RELEASE-NOTES	27/10/2022 13:51	File	7 KB
📄	RUNNING	27/10/2022 13:51	Documento di testo	17 KB

- cartella bin: al suo interno abbiamo una serie di file, ma quelli che ci interessano sono startup.bat e shutdown.bat (in ambiente Windows), startup.sh e shutdown.sh (in ambiente Linux). I file chiamati startup servono per avviare Apache TomEE (cioè il server), mentre i file chiamati shutdown servono per spegnerlo. Se clicchiamo due volte sul file startup, compare una schermata nera che, in caso di avvio del server avvenuto con successo, ci avvisa con un messaggio a video “Server startup in tot ms”. Se, come detto prima, non c’è nessuna variabile d’ambiente JAVA_HOME impostata con un JDK, allora il server non partirà mai, perché non riesce a trovare nessun ambiente JAVA installato. Possiamo fare una verifica anche tramite prompt dei comandi con il comando netstat -a, oppure tramite browser utilizzando l’indirizzo “localhost:8080” :

The screenshot shows two windows side-by-side. The left window is a command prompt titled 'Selezione Tomcat' with the following log output:

```

=C:\Program Files\java\apache-tomee-plume-8.0.13\webapps\ROOT)
27-Oct-2022 14:44:20.900 INFORMAZIONI [main] org.apache.jasper.servlet.TldScanner.scanJars At least one JAR was scanned for TLDs yet
ntained no TLDs. Enable debug logging for this logger for a complete list of JARs that were scanned but no TLDs were found in them.
pping unneeded JARs during scanning can improve startup time and JSP compilation time.
27-Oct-2022 14:44:20.909 INFORMAZIONI [main] sun.reflect.DelegatingMethodAccessorImpl.invoke Deployment of web application directory
:\Program Files\java\apache-tomee-plume-8.0.13\webapps\ROOT] has finished in [77] ms
27-Oct-2022 14:44:20.912 INFORMAZIONI [main] sun.reflect.DelegatingMethodAccessorImpl.invoke Starting ProtocolHandler ["http-nio-8080"]
27-Oct-2022 14:44:20.922 INFORMAZIONI [main] sun.reflect.DelegatingMethodAccessorImpl.invoke Server startup in [799] milliseconds

```

The right window is another command prompt titled 'Selezione Prompt dei comandi' showing the output of the 'netstat -a' command:

```

(c) Microsoft Corporation. Tutti i diritti riservati.

C:\Users\39346>netstat -a

Connessioni attive

Proto Indirizzo locale      Indirizzo esterno      Stato
TCP   0.0.0.0:135           MSI:0                  LISTENING
TCP   0.0.0.0:445           MSI:0                  LISTENING
TCP   0.0.0.0:5040          MSI:0                  LISTENING
TCP   0.0.0.0:8080          MSI:0                  LISTENING
TCP   0.0.0.0:49664          MSI:0                  LISTENING
TCP   0.0.0.0:49665          MSI:0                  LISTENING
TCP   0.0.0.0:49668          MSI:0                  LISTENING
TCP   0.0.0.0:49669          MSI:0                  LISTENING
TCP   0.0.0.0:49676          MSI:0                  LISTENING
TCP   0.0.0.0:49678          MSI:0                  LISTENING
TCP   127.0.0.1:6942         MSI:0                  LISTENING

```

The screenshot shows the Apache Tomcat 9.0.68 welcome page at localhost:8080. The page includes the Apache logo and a message: "If you're seeing this, you've successfully installed Tomcat. Congratulations!". It features a cartoon cat icon and links to recommended reading: Security Considerations How-To, Manager Application How-To, and Clustering/Session Replication How-To. On the right, there are links to Server Status, Manager App, and Host Manager. Below the main message, there's a 'Developer Quick Start' section with links to Tomcat Setup, First Web Application, Realms & AAA, JDBC DataSources, Examples, Servlet Specifications, and Tomcat Versions. The page is divided into three main columns: 'Managing Tomcat', 'Documentation', and 'Getting Help'.

Managing Tomcat

For security, access to the [manager webapp](#) is restricted. Users are defined in: `$CATALINA_HOME/conf/tomcat-users.xml`

In Tomcat 9.0 access to the manager application is split between different users. [Read more...](#)

[Release Notes](#)

[Changelog](#)

[Migration Guide](#)

Documentation

[Tomcat 9.0 Documentation](#)

[Tomcat 9.0 Configuration](#)

[Tomcat Wiki](#)

Find additional important configuration information in: `$CATALINA_HOME RUNNING.txt`

Developers may be interested in:

- [Tomcat 9.0 Bug Database](#)
- [Tomcat 9.0 JavaDocs](#)
- [Tomcat 9.0 Git Repository at GitHub](#)

Getting Help

[FAQ and Mailing Lists](#)

The following mailing lists are available:

- tomcat-announce**: Important announcements, releases, security vulnerability notifications. (Low volume).
- tomcat-users**: User support and discussion.
- taglibs-user**: User support and discussion for [Apache Taglibs](#).
- tomcat-dev**: Development mailing list, including commit messages.

- cartella conf: al suo interno abbiamo una serie di file, ma quello che ci interessa è il file server.xml, che contiene le informazioni relative alle porte e al server utilizzato:

```

61      -->
62
63
64      <!-- A "Connector" represents an endpoint by which requests are received
65          and responses are returned. Documentation at :
66          Java HTTP Connector: /docs/config/http.html
67          Java AJP Connector: /docs/config/ajp.html
68          APR (HTTP/AJP) Connector: /docs/apr.html
69          Define a non-SSL/TLS HTTP/1.1 Connector on port 8080
70      -->
71      <Connector port="8080" protocol="HTTP/1.1"
72          connectionTimeout="20000"
73          redirectPort="8443" xpoweredBy="false" server="Apache TomEE" />
74      <!-- A "Connector" using the shared thread pool -->
75      <!--
76          <Connector executor="tomcatThreadPool"
77              port="8080" protocol="HTTP/1.1"
78              connectionTimeout="20000"
79              redirectPort="8443" />
80      -->
81      <!-- Define an SSL/TLS HTTP/1.1 Connector on port 8443
82          This connector uses the NIO implementation. The default
83          SSL implementation will depend on the presence of the APR/native

```

La porta 8080 del server è possibile anche cambiarla;

- cartella lib: contiene una serie di librerie che implementano i vari livelli delle applicazioni JAVAEE;
- cartella logs: contiene i file di log di default, e può contenere file di log creati da noi per ogni singola applicazione JAVAEE;
- cartella temp: contiene dei file temporanei che vengono rimossi quando il server viene spento;
- cartella webapps: conterrà tutte le nostre applicazioni;
- cartella work: contiene tutti i file .class e .java compilati.

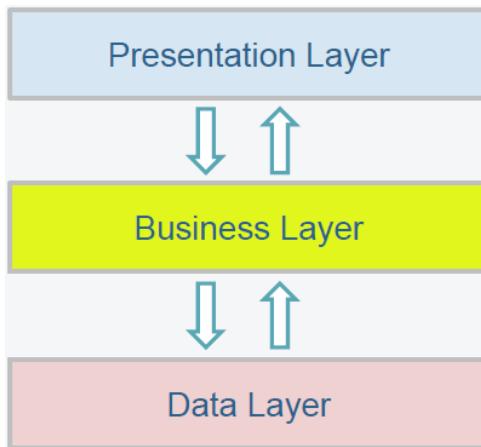
Apache TomEE è basato su 3 software:

1. Catalina: è il modulo che consente di eseguire e interpretare le servlet;
2. Coyote: è il modulo che gestisce tutta la parte relativa al protocollo HTTP, quindi svolge il ruolo di web server;
3. Jasper: è il modulo che consente di eseguire le JSP (Java Server Pages).

WEB APPLICATION

ANATOMIA DI UNA WEB APPLICATION

Una web application generalmente è composta da 3 livelli applicativi:



Questa suddivisione in livelli applicativi costituisce un design pattern, cioè uno schema architettonico che possiamo applicare in più contesti.

PRESENTATION LAYER

Il presentation layer è l'interfaccia utente e si occupa di:

- acquisire i dati inseriti dall'utente generalmente attraverso i form ed altre interazioni dell'utente (click su link, selezioni di opzioni e così via);
- far visualizzare i dati all'utente, attraverso tabelle, grafici, form di modifica dati e così via;

I linguaggi più utilizzati nel presentation layer sono HTML, CSS e Javascript (ed i vari framework).

In JAVA il presentation layer si realizza mediante JSP (JavaServer Pages) e JSF (JavaServer Faces), che contengono appunto HTML, CSS, JavaScript e Scriptlet.

BUSINESS LAYER

Il business layer implementa il Domain Model, cioè tutta la logica applicativa relativa all'accesso e alla gestione dei dati, quindi i servizi.

Il business layer non deve dipendere dall'interfaccia utente e deve poter essere utilizzato da qualsiasi tipo di applicazione (web, client, web service e così via).

In JAVA il business layer si realizza mediante EJB, Entities, Facade e così via.

DATA LAYER

Il data layer si occupa della gestione dell'accesso ai dati presenti sul database e della loro persistenza (gestione della coerenza dei dati all'interno del database).

In particolare, il data layer si occupa di ricevere e gestire le richieste di accesso (in lettura/scrittura) che arrivano dallo strato superiore, ossia il Business Layer.

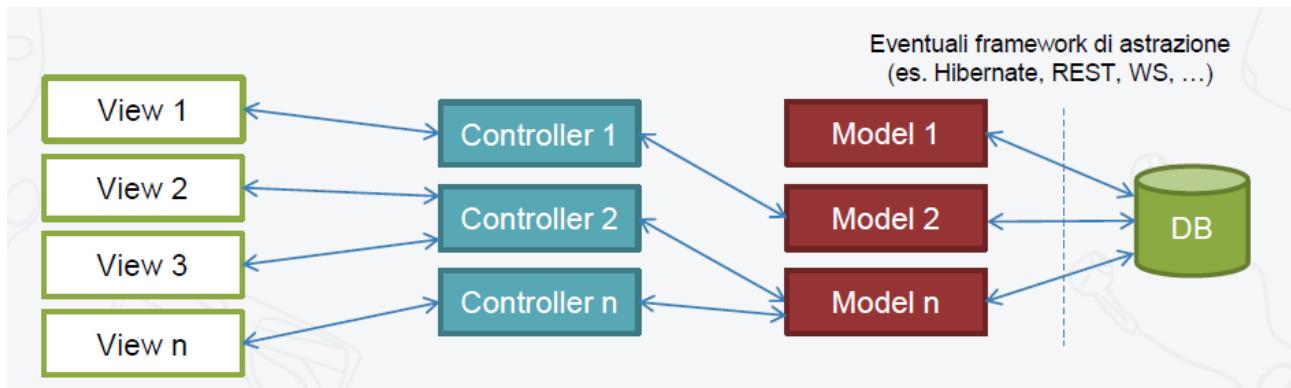
Il data layer contiene classi ed interfacce per implementare il CRUD (Create, Read, Update, Delete). Queste classi sono generalmente utilizzate all'interno del business layer.

Il JDBC è un esempio di data layer, così come Hibernate ed i vari ORM.

PATTERN MODEL-VIEW-CONTROLLER (MVC)

Il pattern MVC è un pattern architettonale che consente di separare nettamente la logica di presentazione dei dati (ovvero le pagine viste dall'utente) dalla logica di business (ovvero le funzionalità per l'accesso a tali dati).

Nel pattern MVC abbiamo tante viste, cioè le pagine HTML, che si interfacciano con i vari controller. I controller a loro volta si interfacciano con la parte model. All'interno del model abbiamo l'interfaccia con il database.



Lato server, il pattern MVC è implementato da numerosi framework:

- JAVA (Spring, JSF e Struts);
- PHP (Symfony, Laravel, CakePHP, ...);
- Python (Django, TurboGears, Pylons, ...).

Lato client, il pattern MVC è implementato da diversi framework di JavaScript (AngularJS, JavascriptMVC, ...).

Il pattern MVC si basa, appunto, sulla separazione netta tra model, view e controller:

- il model implementa tutti i metodi per l'accesso ai dati e si interfaccia con il database, o con eventuali altri livelli di astrazione dei dati (ad es. hibernate, web services, REST services e così via);
- il controller gestisce le richieste di accesso che arrivano alle view (viste), interfacciandosi se necessario con il model per l'accesso ai dati;

- la view gestisce tutte le componenti visualizzate dall'utente (pagine testuali, form, liste, dati recuperati dal model e così via).

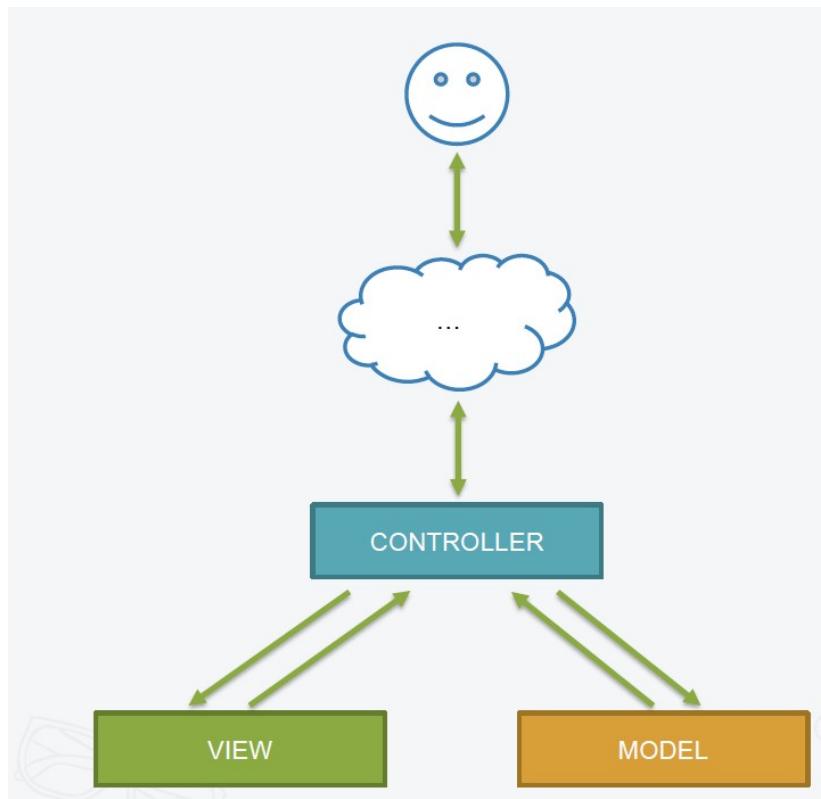
Il funzionamento dell'MVC in una web application è il seguente:

1. un utente effettua una richiesta di accesso all'URL, attraverso un client (browser, app e così via);
2. all'URL risponde la web application che elabora questa richiesta ricevuta.

La web application sviluppata senza pattern MVC sarà fatta da una pagina JSP, ad esempio, che conterrà al suo interno un mix tra HTML e logica applicativa per accedere ai dati presenti sul database, elaborarli e generare l'HTML. Se poi ho un'altra pagina che deve accedere agli stessi dati, devo implementare le stesse funzionalità, per non parlare dei problemi di sicurezza di accesso ai dati.

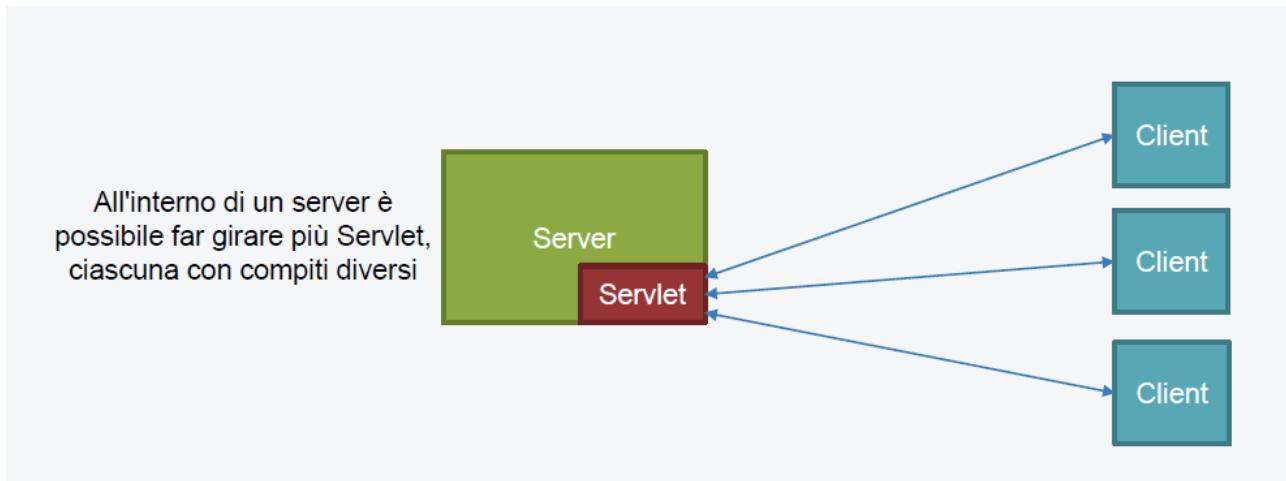
In particolare, le tre parti del pattern MVC per una web application funzionano nel seguente modo:

1. il model si occupa di recuperare e ricevere i dati;
2. la view (la pagina web) che si occuperà di visualizzare l'HTML all'utente;
3. il controller riceve e gestisce la URL richiesta dall'utente e fa da intermediario tra model e view.



SERVLET

Una servlet è un software scritto in JAVA che è in grado di ricevere ed elaborare richieste dal client, quindi una sorta di mini applicazione web.



All'interno di un server possiamo installare più servlet, dove ciascuna può avere un compito diverso.

Una servlet è identificata da un path e più client possono invocare una servlet per effettuare delle richieste e ricevere in output delle risposte. Le risposte possono essere codice HTML, file PDF, file JSON, file XML e così via, a seconda del tipo di servlet che stiamo sviluppando.

Le servlet vengono installate (tecnicamente si dice fare il “deploy”) all’interno di un Servlet Container (ad esempio Apache Tomcat).

Il Servlet Container è una piattaforma software in grado di eseguire applicazioni web sviluppate in JAVA.

Una servlet non ha delle interfacce grafiche, ma è semplicemente una classe JAVA che riceve una richiesta, la elabora e restituisce una risposta, quindi non troviamo librerie SWT, AWT e Swing disponibili per una servlet, in quanto sono librerie dedicate a sviluppo di applicazioni client JAVA.

Lo standard delle servlet rientra all’interno di un insieme di standard, detto JAVA EE.

Generalmente, quando sviluppiamo applicazioni web, non utilizziamo direttamente delle servlet, nel senso che non si usa creare tante servlet per quante sono le funzionalità che dobbiamo implementare, ma si utilizzano framework che implementano la specifica servlet, per esempio Spring, JSF, Struts e così via.

Le vecchie applicazioni web erano composte principalmente da servlet e JavaServer Pages (JSP), mischiando codice HTML con logica applicativa.

Una servlet adesso viene utilizzata per sviluppare un singolo servizio, ad esempio per il download di file PDF.

Il package che contiene tutte le classi e le interfacce relative alle servlet è javax.servlet.

Abbiamo diverse tipologie di servlet, ognuna delle quali estende la classe principale javax.servlet.GenericServlet.

Quando si lavora in ambito web, la classe da utilizzare per la realizzazione di servlet è javax.servlet.http.HttpServlet.

Quindi creare una servlet vuol dire essenzialmente creare una classe JAVA che estende la classe HttpServlet.

L'annotation @WebServlet indica al server che la classe creata è un HttpServlet e richiede come parametro d'ingresso il path da utilizzare per poter essere invocata.

Vediamo un esempio:

```
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;

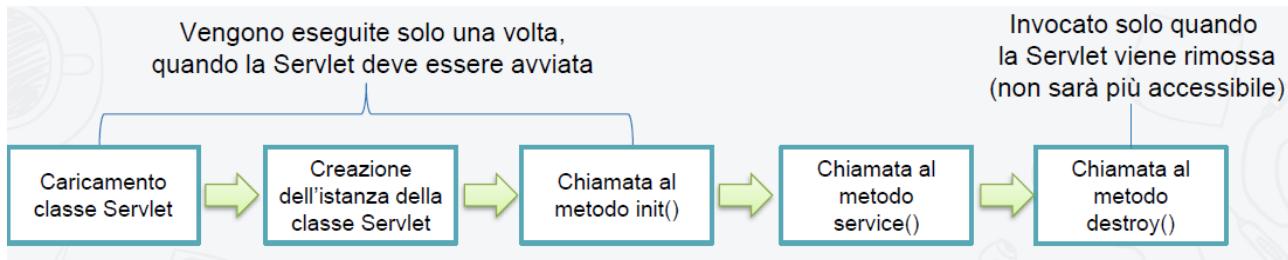
@WebServlet("/downloadpdf")
public class EsempioServlet extends HttpServlet {
```

}

CICLO DI VITA DI UNA SERVLET

Una servlet ha un ciclo di vita ben definito che stabilisce:

- come caricare una servlet in memoria;
- come istanziare un oggetto della classe Servlet creata;
- come inizializzare la servlet;
- come gestire le richieste da parte dei client;
- come disinstallare la servlet;



Il ciclo di vita di una servlet è diviso in 3 gruppi, dove:

1. il caricamento della classe Servlet, la creazione della sua istanza e la chiamata al metodo init() vengono eseguiti una sola volta quando la servlet viene avviata;
2. la chiamata al metodo service() viene effettuata una volta che la servlet viene invocata da parte dei client;
3. la chiamata al metodo destroy() viene effettuata solo quando la servlet deve essere disinstallata.

Il ciclo di vita, appunto, è gestito attraverso i tre metodi dell'interfaccia javax.servlet.Servlet:

- init(ServletConfig config), viene invocato solo quando la servlet viene avviata;
- service(ServletRequest req, ServletResponse res), viene invocato ogni volta che la classe Servlet viene richiamata;

- `destroy()`, viene invocato quando la servlet viene disinstallata, o distrutta.

Tutte le servlet devono implementare questi metodi direttamente, oppure attraverso le classi astratte `GenericServlet` o `HttpServlet`).

I metodi `init()`, `service()` e `destroy()` vengono invocati dal container (per esempio Apache Tomee) per gestire tutto il ciclo di vita della servlet.

Vediamo nel dettaglio tutte le fasi del ciclo di vita di una servlet:

1. Fase 1 e Fase 2 – Caricamento e creazione dell’istanza: in queste due fasi il container carica la classe `Servlet` e crea un oggetto del tipo della nostra classe. Il caricamento può avvenire all’avvio del server, oppure quando viene effettuata la prima richiesta da parte di un client;
2. Fase 3 – Inizializzazione: questa fase si ha dopo aver caricato ovviamente l’istanza (dopo aver creato l’oggetto di tipo `Servlet`), viene eseguita anch’essa dal server e inizializza le risorse di cui ha bisogno la servlet. Per esempio, in questa fase si possono istanziare le connessioni al database, oppure si può verificare che i REST web service da invocare rispondono correttamente e così via;
3. Fase 4 – Gestione delle richieste: è la fase in cui riceviamo la richiesta da parte del client, la elaboriamo e inviamo al client la risposta. La richiesta è rappresentata da un oggetto di tipo `javax.servlet.ServletRequest`, mentre la risposta è rappresentata da un oggetto di tipo `javax.servlet.ServletResponse`. Questi due oggetti sono passati in ingresso al metodo `service()` dell’interfaccia `Servlet` (`service(ServletRequest req, ServletResponse resp)`). Nel caso di richieste web lavoriamo sul protocollo HTTP, per cui gli oggetti non sono di tipo `ServletRequest` e `ServletResponse`, ma di tipo `HttpServletRequest` e `HttpServletResponse`.
4. Fase 5 – Distruzione: in questa fase viene invocato il metodo `destroy()`, che libera le risorse utilizzate dalla servlet e la disinstalla. Per esempio, in questa fase si possono chiudere le connessioni al database o ai servizi, si possono scrivere eventuali log applicativi e così via.

La classe astratta `HttpServlet`, oltre ad implementare i metodi dell’interfaccia `Servlet` (`init()`, `destroy()` e `service()`), implementa anche tutta un’altra serie di metodi:

- `doGet` per gestire richieste HTTP GET;
- `doPost` per gestire richieste HTTP POST;
- `doPut` per gestire richieste HTTP PUT;
- `doDelete` per gestire richieste HTTP DELETE;
- `doHead` per gestire richieste HTTP HEAD;
- `doOptions` per gestire richieste HTTP OPTIONS;
- `doTrace` per gestire richieste HTTP TRACE.

Tutti i metodi prendono in ingresso due oggetti di tipo `HttpServletRequest` e `HttpServletResponse`. I metodi più usati nelle web application sono `doGet` e `doPost`. In altri contesti (ad esempio le API RESTfull) vengono utilizzati anche gli altri.

```

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@WebServlet("/downloadpdf")
public class EsempioServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
    ServletException, IOException {
        resp.getWriter().append("Served at: ").append(req.getContextPath());
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws
    ServletException, IOException {
        super.doPost(req, resp);
    }

    @Override
    public void init() throws ServletException {
        /* istanziare connessioni al DB */
        /* verificare che i REST web service da invocare rispondono correttamente */
        /* etc... */

        super.init();
    }

    @Override
    public void destroy() {
        /* chiudere connessioni al DB o ai servizi */
        /* scrivere eventuali log applicativi */
        /* etc... */

        super.destroy();
    }
}

```

HTTPSERVLETREQUEST E HTTPSERVLETRESPONSE

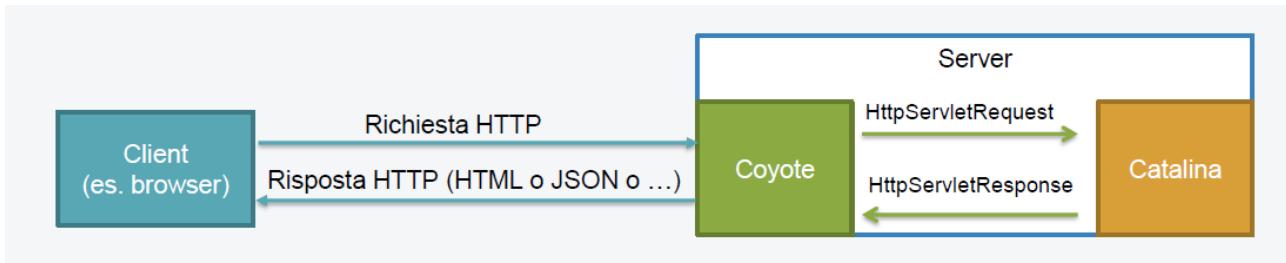
HttpServletRequest e HttpServletResponse sono interfacce contenute nel package javax.servlet.http .

Quando un client chiede l'accesso ad una risorsa web (ad esempio quando scriviamo l'URL di una pagina web nella barra degli indirizzi), lo scenario è il seguente:

- il connettore HTTP (per esempio Coyote per Apache TomEE) riceve la richiesta HTTP, che può essere di tipo GET o POST, e la reindirizza al servlet container (per esempio Catalina per Apache TomEE);
- il servlet container vede il path e a quale servlet è associata. Se la servlet non è caricata, la carica in memoria e la inizializza mediante il metodo init();
- dopo aver inviato la richiesta alla servlet, il servlet container incapsula la richiesta HTTP in un oggetto di tipo HttpServletRequest;
- la servlet elabora la richiesta e scrive la risposta (ad esempio un codice HTML), inserendola poi all'interno di un oggetto di tipo HttpServletResponse;

- la risposta di tipo `HttpServletResponse` viene reindirizzata al web server (ad esempio Coyote per Apache TomEE), che la convertirà per il client che ha effettuato la nostra richiesta.

Le due interfacce `HttpServletRequest` e `HttpServletResponse` sono importanti perché servono per rappresentare a livello logico tutte le informazioni provenienti dalla richiesta e tutte le informazioni che il server invia in risposta. Se non avessimo queste due interfacce, dovremmo utilizzare un array di stringhe, un array di oggetti o una mappa per contenere le informazioni.



I principali metodi dell’interfaccia `HttpServletRequest` sono:

- `request.getMethod()`: restituisce il nome del metodo HTTP utilizzato per effettuare la richiesta al server, che può essere GET, POST, PUT e così via;
- `request.getLocale()`: restituisce la lingua predefinita all’interno del browser;
- `request.getCharacterEncoding()`: restituisce la codifica dei caratteri inviata a livello di request. Le codifiche principali sono ISO 8859-1 e UTF-8;
- `request.getContentType()`: restituisce il MIME type del body della request. I MIME type vengono utilizzati nel protocollo HTTP per codificare i messaggi, quindi definiscono quello che sarà il formato del body della request. Alcuni MIME type che sono utilizzati nello sviluppo di applicazioni web sono:
 - `text/html`: utilizzato per visualizzare contenuti in formato HTML;
 - `multipart/form-data`: utilizzato per indicare che il contenuto della request contiene anche file in upload;
 - `application/pdf`: utilizzato per visualizzare contenuti in formato PDF;
 - `application/vnd.openxmlformats-officedocument.spreadsheetml.sheet`: utilizzato per visualizzare contenuti in formato Excel;
 - `application/json`: utilizzato per visualizzare contenuti in formato JSON;
- `request.getContextPath()`: restituisce la parte della request URI, che corrisponde al path associato alla servlet o, in generale, alla web application invocata. Il path inizia con il simbolo "/" e non termina mai con il simbolo "/" (ad esempio, `/login` può essere il path associato ad una servlet che effettua il login);
- `request.getCookies()`: restituisce un array contenente tutti i cookies inviati con la request. I cookies vengono inseriti all’interno di oggetti di tipo `javax.servlet.http.Cookie`. I cookie sono dei file che gli sviluppatori inseriscono all’interno delle applicazioni per salvare delle informazioni che gli possono essere utili durante la navigazione dell’utente. Vengono usati perché, essendo il protocollo HTTP di tipo stateless (cioè tra una richiesta e l’altra non mantiene informazioni), in questo modo siamo in

grado di gestire gli stati di una web application;

- `request.getHeader(String name)`: restituisce il valore del parametro associato all'header;
- `request.getHeaderNames()`: restituisce un'enumeration di tutti i valori dei parametri presenti nell'header della request;
- `request.getQueryString()`: restituisce la query string contenuta nell'URL. La query string è la stringa che si trova dopo il simbolo << ? >>. Il metodo ritorna null se la URL non contiene query string. Nella query string tutti i parametri sono composti dalla coppia nome = valore. In caso di più parametri nella query string, si usa il simbolo << & >> per separarli. Per esempio:
 - URL senza query string: www.mysite.it/about
 - URL con query string: www.mysite.it/about?param1=abc¶m2=def (la query string è param1=abc¶m2=def)
- `request.getParameter(String name)`: consente di recuperare il valore associato ad un parametro, se il parametro esiste, altrimenti tale metodo restituisce null. I parametri sono quelli che compongono la query string;
- `request.getParameterMap()`: restituisce una mappa che contiene l'elenco dei parametri, con relativi valori, presenti nella request;
- `request.getParameterNames()`: restituisce un enumeration contenente tutti i nomi dei parametri passati nella request;
- `request.getParameterValues(String name)`: restituisce tutti i valori associati ad un parametro all'interno di un array di stringhe, in quanto possiamo specificare per un parametro più valori;
- `request.getAttribute(String name)`: restituisce un oggetto (può essere una stringa, ma anche un oggetto complesso) associato al nome passato in ingresso o null se non esiste un attributo con quel nome. Gli attributi consentono di impostare delle variabili che sono limitate temporaneamente all'esecuzione della request all'interno della nostra servlet o del nostro contesto di web application;
- `request.getAttributeNames()`: restituisce un enumeration contenente i nomi degli attributi presenti nella request;
- `request.getRemoteAddr()`: restituisce l'indirizzo IP del client che ha effettuato la request;
- `request.getRemoteUser()`: restituisce una stringa contenente il parametro utilizzato dall'utente per il login (generalmente l'username o l'email). Il metodo ritorna null se non esiste un utente autenticato;
- `request.getUserPrincipal()`: restituisce un oggetto di tipo `java.security.Principal` che contiene il nome dell'utente autenticato o null se non esiste un utente autenticato. L'interfaccia `Principal` definisce un metodo `getName()` per recuperare il nome associato all'oggetto;
- `request.getRequestURL()`: restituisce un oggetto di tipo `StringBuffer` che contiene tutta la URL della request, compreso il protocollo, il server name, il numero della porta ed il server path. La stringa ritornata non contiene la query string. Per esempio:

URL: <http://miosito.it/miaservlet/param?method=abc>

requestURL(): http://miosito.it/miaservlet/param

- request.getRequestURI(): restituisce la porzione di URL che si trova dopo il server name. Per esempio:

URL: http://miosito.it/miaservlet/param?method=abc

requestURI(): /miaservlet/param

- request.getServerName(): restituisce il nome del server su cui è in esecuzione la web application;
- request.getServerPort(): restituisce il numero della porta del server su cui è in esecuzione la web application;
- request.getServletPath(): restituisce la parte di URL a cui è associata la servlet;
- request.getSession(): restituisce un oggetto di tipo javax.servlet.http.HttpSession contenente la sessione associata alla request ricevuta. Se non esiste una session, questo metodo la crea;
- request.setAttribute(String name, Object value): consente di settare una variabile da utilizzare durante la request. Tutti gli attributi vengono cancellati tra una request e l'altra.

I principali metodi dell'interfaccia HttpServletResponse sono:

- response.addCookie(Cookie arg0): aggiunge un cookie alla response;
- response.addHeader(String name, String value): aggiunge un parametro nell'header della response;
- response.flushBuffer(): forza l'invio del contenuto presente nel buffer al client;
- response.getOutputStream(): restituisce un oggetto di tipo javax.servlet.ServletOutputStream utilizzabile per scrivere dati binary nella response (ad esempio per inviare un PDF al client ...). L'invocazione del metodo flush() sull'oggetto effettua l'invio della response al client;
- response.getWriter(): restituisce un oggetto di tipo java.io.PrintWriter che può essere utilizzato per inviare testo (ad esempio codice HTML) al client;
- response.sendError(int statusCode, String message): invia un error con lo status code indicato al client;
- response.sendRedirect(String URL): è utilizzata per reindirizzare la risposta ad un'altra risorsa (ad esempio una servlet o una JSP). Il parametro da passare in input è una URL relativa o assoluta. Il metodo lavora a livello client perché utilizza la barra degli indirizzi dl browser per effettuare una nuova request (quando si riceve la risposta, nella barra degli indirizzi comparirà la URL dove si verrà reindirizzati);
- response.setStatus(int sc): imposta lo status code della response.

Vediamo un esempio in cui vengono utilizzati alcuni metodi appena spiegati:

```
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.Enumeration;

@WebServlet("/primaservlet") //http://localhost:8080/CORSO_JAVA_EE/primaservlet

public class EsempioServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        azioni(req, resp);

        resp.addCookie(new Cookie("corso-web", "success"));
        resp.getWriter().append("\n - nomeAttributo: " + (String)req.getAttribute("nomeAttributo"));
        // nomeAttributo: valore di prova
    }

    private void azioni(HttpServletRequest req, HttpServletResponse resp) throws IOException {
        resp.getWriter().append("\n - Served at: ").append(req.getContextPath()); // Served at: /CORSO_JAVA_EE
        resp.getWriter().append("\n - Method: " + req.getMethod()); // Method: GET
        resp.getWriter().append("\n - User agent: " + req.getHeader("user-agent"));
        // User agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
        Chrome/106.0.0.0 Safari/537.36

        Enumeration<String> en = req.getHeaderNames();
        while (en.hasMoreElements()){
            String element = en.nextElement();
            resp.getWriter().append("\n - element: " + req.getHeader(element));
            /* - element: localhost:8080
             * - element: keep-alive
             * - element: max-age=0
             * - element: "Chromium";v="106", "Google Chrome";v="106", "Not;A=Brand";v="99"
             * - element: ?0
             * - element: "Windows"
             * - element: 1
             * - element: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
             Chrome/106.0.0.0 Safari/537.36
             * - element: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/
             apng,*;q=0.8,application/signed-exchange;v=b3;q=0.9
             * - element: none
             * - element: navigate
             * - element: ?1
             * - element: document
             * - element: gzip, deflate, br
             * - element: it-IT,it;q=0.9,en-US;q=0.8,en;q=0.7 */
        }

        resp.getWriter().append("\n - Query string: " + req.getQueryString()); // Query string: null
        req.setAttribute("nomeAttributo", "valore di prova");
        resp.getWriter().append("\n - URL: " + req.getRequestURL());
        // URL: http://localhost:8080/CORSO_JAVA_EE/primaservlet
        resp.getWriter().append("\n - URI: " + req.getRequestURI()); // URI: /CORSO_JAVA_EE/primaservlet
        resp.getWriter().append("\n - Servlet: " + req.getServletPath()); // Servlet: /primaservlet
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        super.doPost(req, resp);
    }

    @Override
    public void init() throws ServletException {
        /* istanziare connessioni al DB */
        /* verificare che i REST web service da invocare rispondono correttamente */
        /* etc... */
    }
}
```

```

        super.init();
    }

    @Override
    public void destroy() {
        /* chiudere connessioni al DB o ai servizi */
        /* scrivere eventuali log applicativi */
        /* etc... */

        super.destroy();
    }
}

```

PASSAGGIO DI PARAMETRI AD UNA SERVLET

Esempio di passaggio di parametri ad una servlet in GET e POST tramite un form in file JSP:

home.jsp

```

<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
   pageEncoding="ISO-8859-1"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
        <title>Home</title>
    </head>
<body>
    <p>Form in GET</p>
    <form action="<%=request.getContextPath() + "/servletForm" %>" method="get">
        <label for="nome">Nome</label>
        <input type="text" name="nome" value="">
        <br>
        <label for="cognome">Cognome</label>
        <input type="text" name="cognome" value="">
        <br>
        <button type="submit" name="invia">Invia</button>
    </form>
    <br>
    <p>Form in POST</p>
    <form action="<%=request.getContextPath() + "/servletForm" %>" method="post">
        <label for="nome">Nome</label>
        <input type="text" name="nome" value="">
        <br>
        <label for="cognome">Cognome</label>
        <input type="text" name="cognome" value="">
        <br>
        <button type="submit" name="invia">Invia</button>
    </form>
</body>
</html>

```

ServletForm.java

```
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.PrintWriter;

@WebServlet("/servletForm") //http://localhost:8080/CORSO_JAVA_EE/servlet/form/home.jsp
public class ServletForm extends HttpServlet{
    @Override
    /**
     * Metodo doGet che controlla la ricezione dei due parametri nome e cognome
     * che arrivano dal form creato nel file home.jsp con method = get
     */
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        String nome = req.getParameter("nome");
        String cognome = req.getParameter("cognome");

        PrintWriter output = resp.getWriter();

        output.println("<!DOCTYPE html>");
        output.println("<head><title>Ciao!</title></head>");
        output.println("<body>");
        output.println("<h1>Ciao " + nome + " " + cognome + "</h1>");
        output.println("<p>sono nel doGet...</p>");
        output.println("</body>");
        output.println("</html>");
    }

    /**
     * Metodo doPost che controlla la ricezione dei due parametri nome e cognome
     * che arrivano dal form creato nel file home.jsp con method = post
     */
    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        String nome = req.getParameter("nome");
        String cognome = req.getParameter("cognome");

        PrintWriter output = resp.getWriter();

        output.println("<!DOCTYPE html>");
        output.println("<head><title>Ciao!</title></head>");
        output.println("<body>");
        output.println("<h1>Ciao " + nome + " " + cognome + "</h1>");
        output.println("<p>sono nel doPost...</p>");
        output.println("</body>");
        output.println("</html>");
    }

    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        super.service(req, resp);
    }
}
```

```

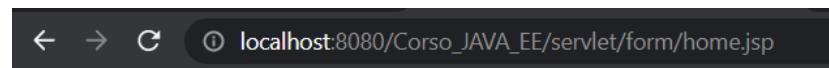
}

@Override
public void destroy() {
    super.destroy();
}

@Override
public void init() throws ServletException {
    super.init();
}
}

```

Il risultato di questo esempio è il seguente:



Form in GET

Nome	<input type="text" value="Paolo"/>
Cognome	<input type="text" value="Preite"/>
<input type="button" value="Invia"/>	

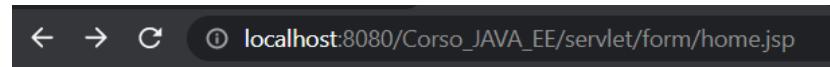
Form in POST

Nome	<input type="text"/>
Cognome	<input type="text"/>
<input type="button" value="Invia"/>	



Ciao Paolo Preite

sono nel doGet...



Form in GET

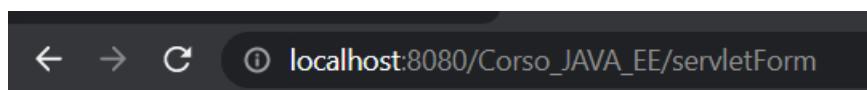
Nome

Cognome

Form in POST

Nome

Cognome



Ciao Paolo Preite

sono nel doPost...

REQUEST DISPATCHING

Quando sviluppiamo una web application, generalmente abbiamo diverse funzionalità e difficilmente riusciamo a realizzare una web application che ha una sola servlet che implementa più funzionalità.

L'attività di dispatching consiste nella gestione della reindirizzazione delle request alla risorsa interessata, che può essere una JSP o un'altra servlet.

Per gestire questa attività di dispatching (attività prevista anche dal paradigma MVC, dove uno o più controller svolgono le attività di elaborazione dei dati e reindirizzazione della request alla JSP o un'altra servlet interessata), possiamo utilizzare l'interfaccia javax.servlet.RequestDispatcher.

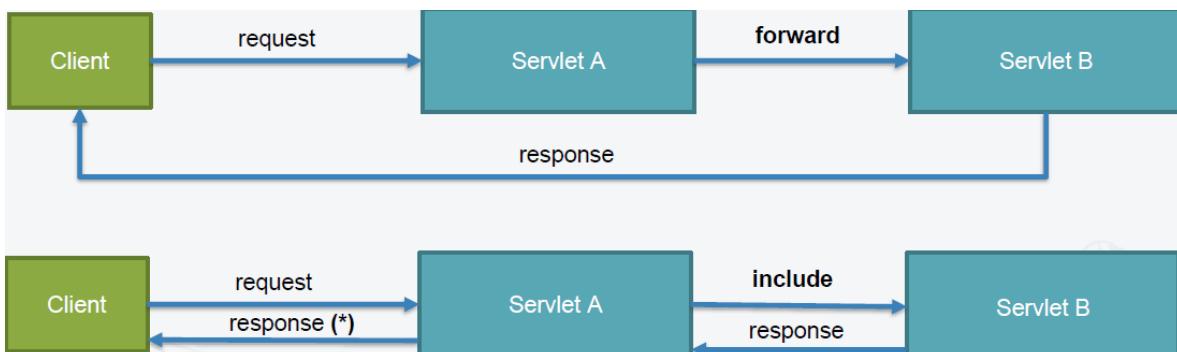
Abbiamo a disposizione due metodi per inoltrare la request:

1. `request.getServletContext().getRequestDispatcher(String jspPath)`, per reindirizzare la request ad una JSP;
2. `request.getServletContext().getNamedDispatcher(String servletName)`, per invocare un'altra servlet;

Entrambi i metodi ritornano un oggetto di tipo RequestDispatcher.

L'interfaccia RequestDispatcher mette a disposizione due metodi:

1. `forward(request, response)`: delega l'invio della risposta alla risorsa a cui ha inoltrato la request;
2. `include(request, response)`: include l'output ottenuto da un'altra risorsa web (ad esempio un'altra servlet) all'interno della nostra risposta.



Vediamo un esempio di più pagine JSP incluse in un'unica servlet:

header.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
        <title>Dispatching</title>
    </head>
<body>
```

body.jsp

```
<h1>Titolo pagina</h1>
<p>
Sono nel body!
</p>
```

footer.jsp

```
</body>
</html>
```

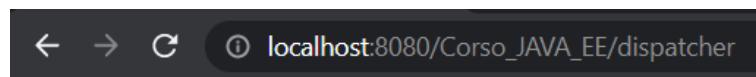
DispatcherServlet.java

```
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@WebServlet("/dispatcher") //http://localhost:8080/CORSO_JAVA_EE/dispatcher
public class DispatcherServlet extends HttpServlet {
    @Override
    /**
     * Metodo doGet che unisce in unico output (in un'unica pagina HTML)
     * i tre file JSP denominati header.jsp, body.jsp e footer.jsp
     */
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException,
    IOException {
        /*Mediante il metodo include stiamo includendo in un'unica response i tre file JSP*/
        req.getServletContext().getRequestDispatcher("/servlet/dispatching/header.jsp").include(req,resp);
        req.getServletContext().getRequestDispatcher("/servlet/dispatching/body.jsp").include(req,resp);
        req.getServletContext().getRequestDispatcher("/servlet/dispatching/footer.jsp").include(req,resp);
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException,
    IOException {
        super.doPost(req, resp);
    }
}
```

Il risultato di questo esempio è il seguente:



Titolo pagina

Sono nel body!

Il dispatching può essere utilizzato anche per gestire le pagine da includere nell'output in base ad un parametro passato alla request, come nell'esempio seguente:

header.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset-ISO-8859-1">
        <title>Dispatching</title>
    </head>
<body>
```

body.jsp

```
<h1>Titolo pagina</h1>
<p>
Sono nel body!
</p>
```

pagina1.jsp

```
<h1>Titolo pagina 1</h1>
<p>
Sono nel body della pagina 1
</p>
```

pagina2.jsp

```
<h1>Titolo pagina 2</h1>
<p>
Sono nel body della pagina 2
</p>
```

pagina3.jsp

```
<h1>Titolo pagina 3</h1>
<p>
Sono nel body della pagina 3
</p>
```

footer.jsp

```
</body>
</html>
```

DispatcherServlet.java

```
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@WebServlet("/dispatcher") //http://localhost:8080/CORSO_JAVA_EE/dispatcher
public class DispatcherServlet extends HttpServlet {
    @Override
    /**
     * Metodo doGet che unisce in unico output (in un'unica pagina HTML)
     * i due file JSP denominati header.jsp e footer.jsp e,
     * in base al parametro inserito nella request, include uno tra i
     * file JSP denominati pagina1.jsp, pagina2.jsp, pagina3.jsp
     * e body.jsp (di default se non ci sono parametri nella request)
     */
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        /*Mediante il metodo include stiamo includendo in un'unica response più file JSP*/
        req.getServletContext().getRequestDispatcher("/servlet/dispatching/header.jsp").include(req,resp);

        String pagina = req.getParameter("pagina");

        /*Controllo se il parametro "pagina" è stato configurato*/
        if (pagina != null && !pagina.trim().equals("")){
            switch (pagina) {
                case "1": //http://localhost:8080/CORSO_JAVA_EE/dispatcher?pagina=1
                    req.getServletContext().getRequestDispatcher("/servlet/dispatching/pagina1.jsp").include(req,resp);
                    break;

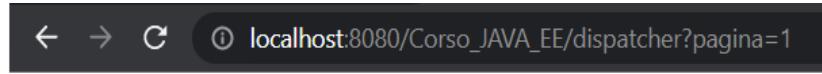
                case "2": //http://localhost:8080/CORSO_JAVA_EE/dispatcher?pagina=2
                    req.getServletContext().getRequestDispatcher("/servlet/dispatching/pagina2.jsp").include(req,resp);
                    break;

                case "3": //http://localhost:8080/CORSO_JAVA_EE/dispatcher?pagina=3
                    req.getServletContext().getRequestDispatcher("/servlet/dispatching/pagina3.jsp").include(req,resp);
                    break;
            }
        } else {
            // http://localhost:8080/CORSO_JAVA_EE/dispatcher
            req.getServletContext().getRequestDispatcher("/servlet/dispatching/body.jsp").include(req,resp);
        }
        req.getServletContext().getRequestDispatcher("/servlet/dispatching/footer.jsp").include(req,resp);
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        super.doPost(req, resp);
    }
}
```

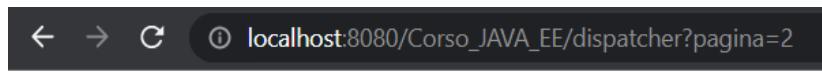
Il risultato di questo esempio è il seguente:





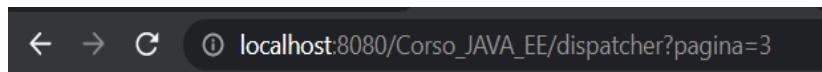
Titolo pagina 1

Sono nel body della pagina 1



Titolo pagina 2

Sono nel body della pagina 2



Titolo pagina 3

Sono nel body della pagina 3

HTTPSESSION

Come abbiamo già potuto specificare nei paragrafi precedenti, il protocollo HTTP è un protocollo stateless. Il termine stateless vuol dire che, tra una richiesta e l'altra, il protocollo HTTP non mantiene informazioni sullo stato delle richieste precedenti, quindi è un protocollo senza memoria.

Questo vuol dire che, quando il client effettua una richiesta, il server restituisce la risposta e chiude la connessione. Nella richiesta successiva il server non sa che il client ha effettuato una richiesta precedente.



Per sopperire a questa mancanza del protocollo HTTP, i web server devono implementare un meccanismo in grado di mantenere traccia delle richieste effettuate dai client. Per esempio, se non ci fossero i mezzi per sopperire a questo limite, in un sito di ecommerce non sarebbe possibile mantenere gli articoli inseriti nel carrello elettronico durante la navigazione del sito stesso.

Per gestire le informazioni tra una richiesta e l'altra, i web server utilizzano un meccanismo che si chiama sessione HTTP.

La sessione HTTP in JAVA viene gestita attraverso due componenti:

- un parametro che si chiama JSESSIONID salvato all'interno di un cookie, che rappresenta l'ID univoco di una sessione. Per cui, se abbiamo due client che fanno una richiesta alla stessa web application, il container imposterà un JSESSIONID per ogni client;
- l'interfaccia javax.servlet.http.HttpSession.

Quando una web application JAVA riceve una richiesta dal client, il Servlet Engine (Catalina nel caso di Apache TomEE) controlla se la request contiene un cookie che ha come nome il JSESSIONID. Se esiste, vuol dire che abbiamo già creato una sessione per quel client, altrimenti viene creata una nuova sessione (viene creato un nuovo oggetto) di tipo HttpSession.

Quindi, a livello di server avremo tanti oggetti di tipo HttpSession per quante saranno le richieste effettuate dai vari client.

La sessione può avere durata limitata o illimitata, a seconda dei vari casi.

Nell'oggetto HttpSession possiamo salvare tutti i dati che provengono dal client.



Nell'esempio del carrello elettronico, quando aggiungiamo un articolo al carrello, inviamo una richiesta al server. Il server, in base al nostro JSESSIONID, prende l'oggetto HttpSession associato a noi e all'interno di tale oggetto ci sarà una variabile (impostata da noi) che conterrà la lista degli articoli che abbiamo aggiunto.

La gestione della sessione si pone al di sopra del protocollo HTTP, quindi non intacca la logica stateless del protocollo HTTP. Infatti, al termine di ogni richiesta, il server chiude sempre e comunque la connessione con il client.

Se i cookie sono disabilitati, possiamo utilizzare un meccanismo che si chiama URL encoding, che consente di riscrivere le URL in modo che contengano al loro interno anche il JSESSIONID. In pratica, mediante l'URL encoding, il JSESSIONID viene passato come parametro all'URL tramite query string.

Per recuperare l'oggetto HttpSession, l'interfaccia HttpServletRequest mette a disposizione il metodo getSession().

```
HttpSession session = request.getSession();
```

Il metodo getSession() restituisce l'oggetto di tipo HttpSession se esiste una sessione attiva, altrimenti ne crea una nuova.

I principali metodi utilizzati sono:

- `getId()`: restituisce il JSESSIONID associato alla sessione;
- `setAttribute(String name, Object value)`: consente di settare una variabile da utilizzare durante la sessione, associandola al nome specificato;
- `getAttribute(String name)`: restituisce la variabile associata al nome indicato o null se non esiste alcun oggetto;
- `getAttributeNames()`: restituisce un enumeration che contiene i nomi di tutti gli attributi contenuti nell'oggetto HttpSession;
- `removeAttribute(String name)`: rimuove un attributo dalla sessione se esiste, altrimenti non fa nulla;
- `getLastAccessedTime()`: restituisce l'ultimo istante temporale (il timestamp...) in cui il client ha inviato una request, espresso in millisecondi. Questo metodo è importante perché viene utilizzato per determinare se la sessione deve rimanere attiva o meno dopo un determinato periodo di tempo. Per esempio, sul nostro profilo del sito della banca, la sessione ha una durata di circa 15

minuti. Superato quel periodo di tempo di inattività sul nostro profilo della banca, la sessione viene terminata e noi siamo costretti a reinserire le credenziali per effettuare di nuovo il login;

- `setMaxInactiveInterval(int interval)`: consente di impostare i secondi di inattività trascorsi i quali la sessione viene terminata. Il valore di default è di 1800 secondi (30 minuti). Se interval è negativo, la sessione sarà sempre attiva, a meno che non invochiamo manualmente il metodo `invalidate()`;
- `getMaxInactiveInterval()`: restituisce il valore impostato tramite `setMaxInactiveInterval(int interval)`.

Vediamo un esempio in cui costruiamo un carrello per inserire gli articoli da acquistare:

carrello.jsp

```
<%@ page import="java.util.* , java.io.*" %>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset-ISO-8859-1">
        <title>Carrello</title>
    </head>
    <body>
        <h1>Carrello</h1>

        <%
            List<String> carrello = (List<String>)
request.getSession().getAttribute("carrello");

            if(carrello != null && carrello.size() > 0) {
                for(String articolo : carrello) {
                    out.println(articolo + "<br>");
                }
            } else {
                out.println("Non ci sono articoli nel carrello!");
            }
        %
    </body>
</html>
```

Carrello.java

```
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

@WebServlet("/carrello") //http://localhost:8080/CORSO_JAVA_EE/carrello
public class Carrello extends HttpServlet{
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException,
IOException {
        List<String> carrello = (List<String>) req.getSession().getAttribute("carrello");
```

```

/*Se il carrello non c'è, lo creiamo*/
if (carrello == null){
    carrello = new ArrayList<>();

    /*Associamo alla variabile "carrello" l'ipotetico carrello presente in memoria.
     * Questa variabile "carrello" è una variabile associata in sessione, per cui quando
     * la sessione si chiude, vengono cancellate tutte le variabili associate ad essa*/
    req.getSession().setAttribute("carrello", carrello);
}

/*Creiamo il parametro dove inserire l'articolo da comprare*/
String articolo = req.getParameter("articolo");

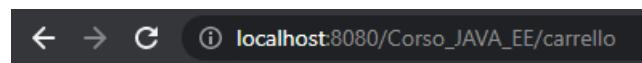
if (articolo != null && !articolo.trim().equals("articolo")) {
    carrello.add(articolo);
}

req.getServletContext().getRequestDispatcher("/servlet/carrello/carrello.jsp").include(req,resp);
}

@Override
protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException,
IOException {
    super.doPost(req, resp);
}
}

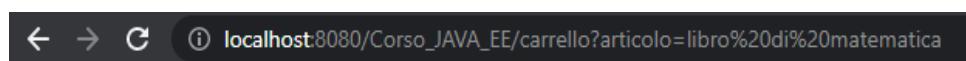
```

Il risultato di questo esempio è il seguente:



Carrello

Non ci sono articoli nel carrello!



Carrello

libro di matematica



Carrello

libro di matematica
libro di italiano



Carrello

[libro di matematica](#)
[libro di italiano](#)
[libro di storia](#)

Da notare nell'esempio che l'attributo “carrello” è stato settato utilizzando il getSession(). Questo perché, se avessimo settato tale attributo senza il getSession() (quindi non a livello di sessione, ma solo a livello di chiamata), ogni articolo inserito nel carrello avrebbe sovrascritto quello precedente, facendo vedere a video solo l'articolo corrente inserito in quel momento. Per cui, se tra una richiesta e l'altra nella stessa sessione vogliamo memorizzare dei valori all'interno di una variabile settata come attributo, tale variabile deve essere sempre impostata come attributo di sessione mediante il getSession(), altrimenti tra una chiamata e l'altra si perdono i dati della chiamata precedente.

JAVA FILTERS E FILTER CHAIN

I filtri sono degli oggetti di tipo `javax.servlet.Filter` e rappresentano uno strumento molto potente, messo a disposizione da JAVA, per la gestione di pre-processing delle richieste e post-processing delle risposte. Il termine pre-processing vuol dire intercettare la richiesta ed effettuare delle operazioni, mentre post-processing vuol dire intervenire subito dopo la risposta ed effettuare delle operazioni.

I filtri intervengono, pertanto, prima che una richiesta raggiunga la servlet o appena dopo che la risposta esca dalla servlet.

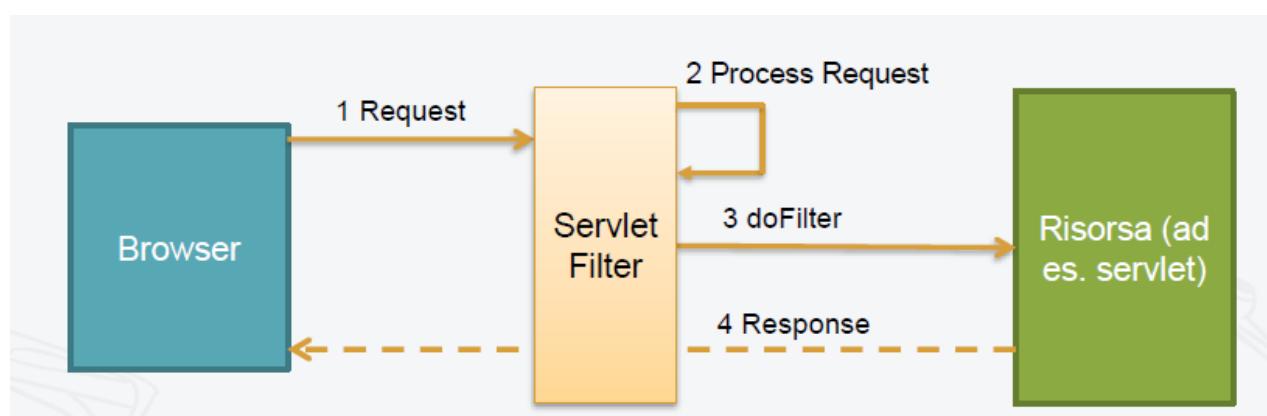
Servono per esempio a:

- verificare se un utente è autenticato. In caso di mancata autenticazione, si termina la sessione e si reindirizza l'utente nella pagina di login;
- filtrare i log;
- comprimere i dati;
- ...

PRE-PROCESSING DELLE RICHIESTE

Il pre-processing delle richieste funziona nel seguente modo:

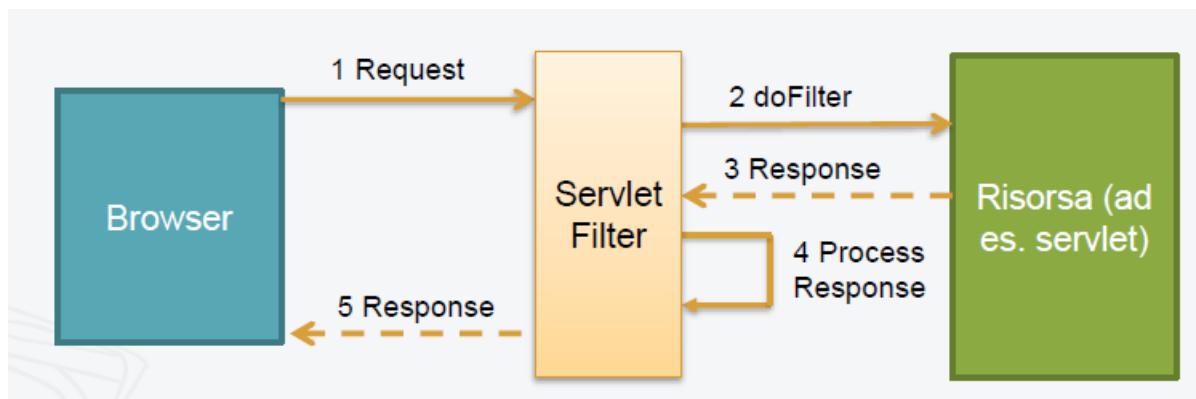
- il client effettua una richiesta al server, quindi digita una URL per invocare la nostra web application o servlet;
- il filtro intercetta la richiesta;
- il filtro effettua una serie di operazioni che pre-processano la richiesta (in questa fase può effettuare accesso al database, accedere ai parametri della request e così via);
- il filtro richiama il metodo `doFilter()` per inoltrare la request alla servlet o, in generale, alla risorsa richiesta;
- la risorsa invocata elabora la richiesta e risponde al client.



POST-PROCESSING DELLE RISPOSTE

Il post-processing delle risposte funziona nel seguente modo:

- il client effettua una richiesta al server, quindi digita una URL per invocare la nostra web application o servlet;
- il filtro intercetta la richiesta;
- il filtro richiama il metodo doFilter() per inoltrare la request alla servlet o, in generale, alla risorsa richiesta;
- la risorsa invocata elabora la richiesta e risponde al client;
- il filtro effettua una serie di operazioni che post-processano la risposta (in questa fase è possibile comprimere i dati, o criptarli e così via);
- la risposta viene inviata al client.



IMPLEMENTAZIONE DI UN FILTRO

Per implementare un filtro in JAVA:

- si crea una classe che implementa l'interfaccia javax.servlet.Filter. È buona norma rinominare tale classe con il suffisso Filter, in modo tale da far capire che questa classe è un filtro;
- si utilizza l'annotation @WebFilter(), che può avere i seguenti parametri in ingresso a seconda delle necessità:
 - “/*” : in questo caso il filtro interviene prima di tutte le servlet, file JSP e web application invocate dal client;
 - “/urlRisorsa”: in questo caso il filtro interviene prima della servlet, file JSP o web application invocata dal client;

All'interno del filtro invece impostiamo i seguenti metodi, che sono definiti nell'interfaccia javax.servlet.Filter:

- il metodo init(FilterConfig filterConfig), invocato quando il filtro viene avviato;
- il metodo doFilter(ServletRequest request, ServletResponse response, FilterChain chain) al cui interno inseriamo la logica applicativa
- il metodo destroy(), invocato quando il filtro viene distrutto;

Vediamo un esempio in cui filtriemo l'articolo “bomba” nel carrello, in modo da non farlo inserire al suo interno:

carrello.jsp

```
<%@ page import="java.util.*, java.io.*" %>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset-ISO-8859-1">
        <title>Carrello</title>
    </head>
    <body>
        <h1>Carrello</h1>

        <%
            List<String> carrello = (List<String>) request.getSession().getAttribute("carrello");

            if(carrello != null && carrello.size() > 0) {
                for(String articolo : carrello) {
                    out.println(articolo + "<br>");
                }
            } else {
                out.println("Non ci sono articoli nel carrello!");
            }
        %>
    </body>
</html>
```

ServletFilter.java

```
import javax.servlet.*;
import javax.servlet.annotation.WebFilter;
import javax.servlet.http.HttpServletRequest;
import java.io.IOException;

/*Il parametro in ingresso di @WebFilter deve essere
 * L'URL della servlet, o file JSP, o web application
 * che la classe con tale annotation deve filtrare.
 * In alternativa il filtro può agire prima di tutti gli URL
 * contemporaneamente con il parametro in ingresso "/*" */
@WebFilter("/carrellofiltrato")
public class ServletFilter implements Filter {
    @Override
    public void init(FilterConfig filterConfig) throws ServletException {

    }

    @Override
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
throws IOException, ServletException {
```

```

HttpServletRequest req = (HttpServletRequest) request;
req.setAttribute("articoloDaScartare", "bomba");
/*Questa riga di codice deve essere sempre presente
nel metodo doFilter e deve essere sempre l'ultima riga*/
chain.doFilter(request, response);
}

@Override
public void destroy() {
}
}

```

CarrelloFiltrato.java

```

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

@WebServlet("/carrellofiltrato") //http://localhost:8080/CORSO_JAVA_EE/carrellofiltrato
public class CarrelloFiltrato extends HttpServlet{
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException,
    IOException {
        List<String> carrello = (List<String>) req.getSession().getAttribute("carrello");

        /*Se il carrello non c'è, lo creiamo*/
        if (carrello == null){
            carrello = new ArrayList<>();

            /*Associamo alla variabile "carrello" l'ipotetico carrello presente in memoria.
             * Questa variabile "carrello" è una variabile associata in sessione, per cui quando
             * La sessione si chiude, vengono cancellate tutte le variabili associate ad essa*/
            req.getSession().setAttribute("carrello", carrello);
        }

        /*Creiamo il parametro dove inserire l'articolo da comprare*/
        String articolo = req.getParameter("articolo");

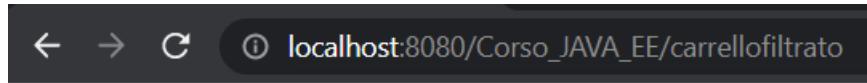
        if (articolo != null &&
            !articolo.trim().equals("articolo") &&
            !articolo.trim().equals(req.getAttribute("articoloDaScartare"))){
            carrello.add(articolo);
        }

        req.getServletContext().getRequestDispatcher("/servlet/carrello/carrello.jsp").include(req,resp);
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException,
    IOException {
        super.doPost(req, resp);
    }
}

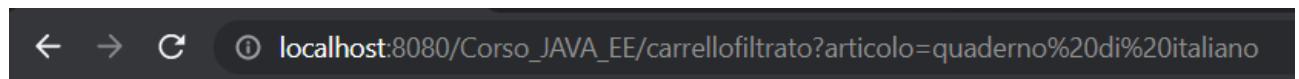
```

Il risultato di questo esempio è il seguente:



Carrello

Non ci sono articoli nel carrello!



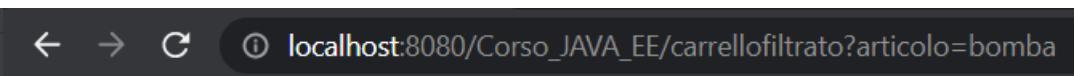
Carrello

quaderno di italiano



Carrello

quaderno di italiano
quaderno di storia



Carrello

quaderno di italiano
quaderno di storia

FILTER CHAIN

Il concetto di filter chain consiste nell'eseguire più filtri in una sequenza prefissata.

Vediamo un esempio in cui applichiamo due filtri sugli articoli pistola e bomba, in modo da non farli inserire all'interno del carrello:

carrello.jsp

```
<%@ page import="java.util.* , java.io.*" %>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset-ISO-8859-1">
        <title>Carrello</title>
    </head>
    <body>
        <h1>Carrello</h1>

        <%
            List<String> carrello = (List<String>) request.getSession().getAttribute("carrello");

            if(carrello != null && carrello.size() > 0) {
                for(String articolo : carrello) {
                    out.println(articolo + "<br>");
                }
            } else {
                out.println("Non ci sono articoli nel carrello!");
            }
        %>
    </body>
</html>
```

CarrelloFiltrato.java

```
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

@WebServlet("/carrellofiltrato") //http://localhost:8080/CORSO_JAVA_EE/carrellofiltrato
public class CarrelloFiltrato extends HttpServlet{
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException,
    IOException {

        List<String> carrello = (List<String>) req.getSession().getAttribute("carrello");

        /*Se il carrello non c'è, lo creiamo*/
        if (carrello == null){
            carrello = new ArrayList<>();

            /*Associamo alla variabile "carrello" l'ipotetico carrello presente in memoria.
             * Questa variabile "carrello" è una variabile associata in sessione, per cui quando
             * La sessione si chiude, vengono cancellate tutte le variabili associate ad essa*/
            req.getSession().setAttribute("carrello", carrello);
        }

        /*Creiamo il parametro dove inserire l'articolo da comprare*/
        String articolo = req.getParameter("articolo");
```

```

        if (articolo != null &&
            !articolo.trim().equals("articolo") &&
            !articolo.trim().equals(req.getAttribute("articoloBomba")) &&
            !articolo.trim().equals(req.getAttribute("articoloPistola")) ) {
            carrello.add(articolo);
        }

        req.getServletContext().getRequestDispatcher("/servlet/carrello/carrello.jsp").include(req,resp);
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException,
    IOException {
        super.doPost(req, resp);
    }
}

```

GodFilter.java

```

import javax.servlet.*;
import javax.servlet.annotation.WebFilter;
import javax.servlet.http.HttpServletRequest;
import java.io.IOException;
import java.util.LinkedHashMap;
import java.util.Map;

@WebFilter("/*")
/**
 * Classe che serve per impostare l'ordine in cui devono essere eseguiti i filtri
 */
public class GodFilter implements Filter {

    private Map<Pattern, Filter> filters = new LinkedHashMap<Pattern, Filter>();

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
        /*Filter1 filter1 = new Filter1();
        filter1.init(config);
        filters.put(new Pattern("/foo/*"), filter1);

        Filter2 filter2 = new Filter2();
        filter2.init(config);
        filters.put(new Pattern("*.bar"), filter2);

        // ...*/

        BombaFilter bombaFilter = new BombaFilter();
        bombaFilter.init(filterConfig);
        filters.put(new Pattern("/carrellofiltrato/*"), bombaFilter);

        PistolaFilter pistolaFilter = new PistolaFilter();
        pistolaFilter.init(filterConfig);
        filters.put(new Pattern("/carrellofiltrato/*"), pistolaFilter);
    }

    @Override
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws
    IOException, ServletException {
        HttpServletRequest hsr = (HttpServletRequest) request;
        String path = hsr.getRequestURI().substring(hsr.getContextPath().length());
        GodFilterChain godChain = new GodFilterChain(chain);

        for (Map.Entry<Pattern, Filter> entry : filters.entrySet()) {
            if (entry.getKey().matches(path)) {
                godChain.addFilter(entry.getValue());
            }
        }

        godChain.doFilter(request, response);
    }
}

```

```

@Override
public void destroy() {
    for (Filter filter : filters.values()) {
        filter.destroy();
    }
}
}

```

Pattern.java

```

public class Pattern {
    private int position;
    private String url;

    public Pattern(String url) {
        this.position = url.startsWith("*") ? 1
            : url.endsWith("*") ? -1
            : 0;
        this.url = url.replaceAll("/?\\/*", "");
    }

    public boolean matches(String path) {
        return (position == -1) ? path.startsWith(url)
            : (position == 1) ? path.endsWith(url)
            : path.equals(url);
    }
}

```

GodFilterChain.java

```

import javax.servlet.*;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class GodFilterChain implements FilterChain {
    private FilterChain chain;
    private List<Filter> filters = new ArrayList<Filter>();
    private Iterator<Filter> iterator;

    public GodFilterChain(FilterChain chain) {
        this.chain = chain;
    }

    @Override
    public void doFilter(ServletRequest request, ServletResponse response) throws
IOException, ServletException {
        if (iterator == null) {
            iterator = filters.iterator();
        }

        if (iterator.hasNext()) {
            iterator.next().doFilter(request, response, this);
        } else {
    }
}

```

```

        chain.doFilter(request, response);
    }
}

public void addFilter(Filter filter) {
    if (iterator != null) {
        throw new IllegalStateException();
    }
    filters.add(filter);
}
}

```

BombaFilter.java

```

import javax.servlet.*;
import javax.servlet.http.HttpServletRequest;
import java.io.IOException;

public class BombaFilter implements Filter {
    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
    }

    @Override
    public void doFilter(HttpServletRequest request, HttpServletResponse response, FilterChain
chain) throws IOException, ServletException {
        HttpServletRequest req = (HttpServletRequest) request;
        req.setAttribute("articoloBomba", "bomba");

        /*Questa riga di codice deve essere sempre presente
        nel metodo doFilter e deve essere sempre l'ultima riga*/
        chain.doFilter(request, response);
    }

    @Override
    public void destroy() {
    }
}

```

PistolaFilter.java

```

import javax.servlet.*;
import javax.servlet.annotation.WebFilter;
import javax.servlet.http.HttpServletRequest;
import java.io.IOException;

public class PistolaFilter implements Filter {
    @Override
    public void init(FilterConfig filterConfig) throws ServletException {

```

```

}

@Override
public void doFilter(ServletRequest request, ServletResponse response, FilterChain
chain) throws IOException, ServletException {
    HttpServletRequest req = (HttpServletRequest) request;
    req.setAttribute("articoloPistola", "pistola");

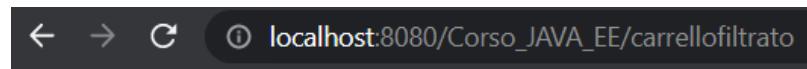
    /*Questa riga di codice deve essere sempre presente
    nel metodo doFilter e deve essere sempre l'ultima riga*/
    chain.doFilter(request, response);
}

@Override
public void destroy() {

}
}

```

Il risultato di questo esempio è il seguente:



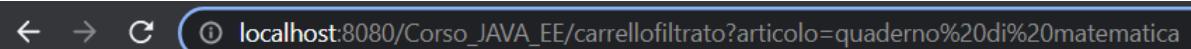
Carrello

Non ci sono articoli nel carrello!



Carrello

quaderno di italiano



Carrello

quaderno di italiano
quaderno di matematica

Carrello

quaderno di italiano
quaderno di matematica

Carrello

quaderno di italiano
quaderno di matematica

SERVLETLISTENERS

I ServletListeners sono oggetti JAVA che sono in ascolto su un particolare evento. Quando si verifica questo evento, i ServletListeners intervengono per eseguire una serie di azioni che abbiamo opportunamente configurato e scritto all'interno della classe.

Gli eventi possono essere l'aggiunta di un attributo alla request, l'inizializzazione di una servlet, il destroy di una servlet e così via.

Abbiamo 8 tipi di listener disponibili:

- **ServletContextListener:** è in ascolto sull'evento SessionContextEvent e notifica quando una servlet viene inizializzata, o distrutta. ServletContextListener è un'interfaccia che definisce due metodi:
 - contextDestroyed(ServletContextEvent e), eseguito quando l'applicazione viene distrutta;
 - contextInitialized(ServletContextEvent e), eseguito quando l'applicazione viene inizializzata;
- **ServletContextAttributeListener:** è in ascolto sull'evento SessionContextAttributeEvent e notifica quando un oggetto viene aggiunto, rimosso o sostituito dalla servlet a livello di applicazione. ServletContextAttributeListener è un'interfaccia che definisce tre metodi:
 - attributeAdded(ServletContextAttributeEvent e), eseguito quando un nuovo attributo viene aggiunto al servlet context;
 - attributeRemoved(ServletContextAttributeEvent e), eseguito quando un attributo viene rimosso dal servlet context;
 - attributeReplaced(ServletContextAttributeEvent e), eseguito quando un attributo viene sostituito nel servlet context;
- **HttpSessionListener:** è in ascolto sull'evento HttpSessionEvent e notifica quando la sessione viene creata, o distrutta. HttpSessionListener è un'interfaccia che definisce due metodi:
 - sessionDestroyed(HttpSessionEvent e), eseguito quando la sessione viene distrutta;
 - sessionCreated(HttpSessionEvent e), eseguito quando la sessione viene creata;
- **HttpSessionAttributeListener:** è in ascolto sull'evento HttpSessionBindingEvent e notifica quando un oggetto viene aggiunto, rimosso o sostituito dalla sessione. HttpSessionAttributeListener è un'interfaccia che definisce tre metodi:
 - attributeAdded(HttpSessionBindingEvent e), eseguito quando un attributo viene aggiunto alla sessione;
 - attributeRemoved(HttpSessionBindingEvent e), eseguito quando un attributo viene rimosso dalla sessione;
 - attributeReplaced(HttpSessionBindingEvent e), eseguito quando un attributo viene sostituito nella sessione;
- **ServletRequestListener:** è in ascolto sull'evento ServletRequestEvent e notifica quando viene creata, o distrutta una request. ServletRequestListener è un'interfaccia che definisce due metodi:
 - requestDestroyed(ServletRequestEvent e), eseguito quando la richiesta viene distrutta;

- `requestInitialized(ServletRequestEvent e)`, eseguito quando la richiesta viene inizializzata;
- `ServletRequestAttributeListener`: è in ascolto sull'evento `ServletRequestAttributeEvent` e notifica quando un oggetto viene aggiunto, rimosso, o sostituito a livello di request.
`ServletRequestAttributeListener` è un'interfaccia che definisce tre metodi:
 - `attributeAdded(ServletRequestAttributeEvent e)`, eseguito quando viene aggiunto un attributo alla request;
 - `attributeRemoved(ServletRequestAttributeEvent e)`, eseguito quando un attributo viene rimosso dalla request;
 - `attributeReplaced(ServletRequestAttributeEvent e)`, eseguito quando un attributo viene sostituito sulla request;
- `HttpSessionActivationListener`, ascolta una sessione e intercetta quando l'oggetto migra da una macchina virtuale all'altra. Questo listener è importante quando siamo ad esempio in un cluster di server. Un cluster di server consiste in più server installati su macchine diverse, ma che sono collegati in qualche modo tra loro e che si scambiano informazioni. Attraverso questo listener possiamo intercettare quando uno dei server sta per spegnersi e possiamo inoltrare la sessione ad un altro server, in modo che l'utente non si accorga che uno dei server non risponde più;
- `HttpSessionBindingListener`, usato per notificare ad un oggetto che è stato aggiunto o rimosso dalla sessione.

Vediamo un esempio di creazione di listener:

AttributoRequestListener.java

```
import javax.servlet.ServletRequestAttributeEvent;
import javax.servlet.ServletRequestAttributeListener;
import javax.annotation.WebListener;

@WebListener
public class AttributoRequestListener implements ServletRequestAttributeListener {
    @Override
    public void attributeAdded(ServletRequestAttributeEvent srae) {
        System.out.println("Oggetto " + srae.getName() + " aggiunto alla request ");
    }

    @Override
    public void attributeRemoved(ServletRequestAttributeEvent srae) {
        System.out.println("Oggetto " + srae.getName() + " cancellato dalla request ");
    }

    @Override
    public void attributeReplaced(ServletRequestAttributeEvent srae) {
        System.out.println("Oggetto " + srae.getName() + " modificato dalla request ");
    }
}
```

SERVLET CHE GENERA PDF CON LA LIBRERIA ITEXT

Itext è una libreria esterna di JAVA che si occupa di generare file pdf e non solo. Essendo una libreria esterna, va scaricata e installata prima di poterla utilizzare.

Vediamo come costruire la servlet che genera pdf con Itext:

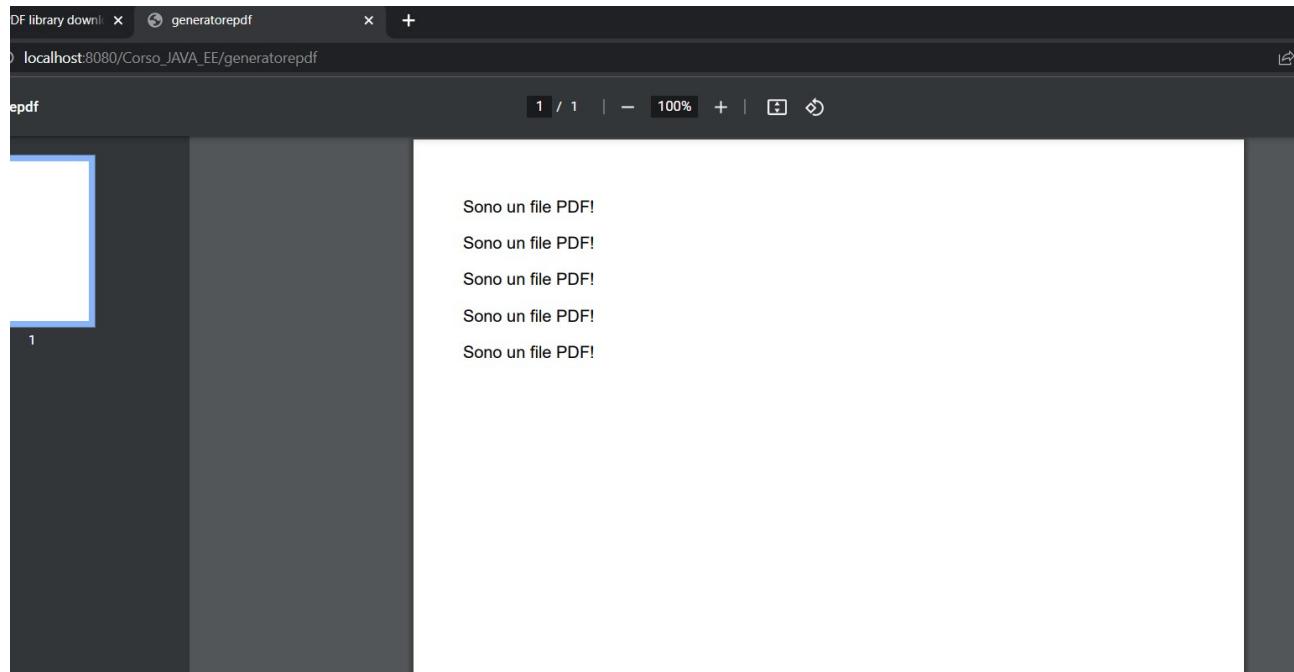
GeneratorePdfServlet.java

```
import com.itextpdf.kernel.pdf.PdfDocument;
import com.itextpdf.kernel.pdf.PdfWriter;
import com.itextpdf.layout.Document;
import com.itextpdf.layout.element.Paragraph;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@WebServlet("/generatorepdf") //http://localhost:8080/CORSO_JAVA_EE/generatorepdf
public class GeneratorePdfServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
    ServletException, IOException {
        PdfWriter pdf = new PdfWriter(resp.getOutputStream());
        PdfDocument doc = new PdfDocument(pdf);
        Document document = new Document(doc);
        document.add(new Paragraph("Sono un file PDF!"));
        document.close();
        resp.setContentType("application/pdf");
        resp.setHeader("Content-disposition", "attachment; filename=corsojava.pdf");
    }
    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws
    ServletException, IOException {
        super.doPost(req, resp);
    }
}
```

Il risultato di questo esempio è il seguente:



La libreria Itext è molto utilizzata per generare file pdf al volo e offre parecchie istruzioni per strutturare il pdf a nostro piacimento. È consigliato andare a studiare questa libreria per vedere cosa ha da offrire.

JAVA SERVER PAGES (JSP)

Le Java Server Pages (JSP) sono una tecnologia importantissima nello sviluppo di applicazioni web.

Le JSP possono essere utilizzate in maniera autonoma per creare applicazioni web, oppure all'interno di framework (es esempio Struts, o Spring).

Le JSP costituiscono la tecnologia di base per la parte view di qualsiasi framework JAVA, sono un mix tra codice HTML e codice JAVA e hanno il vantaggio di essere pagine dinamiche.

Un file JSP ha un'estensione .jsp e tutto il codice JAVA viene racchiuso tra i simboli <% e %> .

Pagina HTML

```
<html>
<body>
  <h1>Titolo</h1>
  <p>Testo della pagina...</p>
</body>
</html>
```

Pagina JSP

```
<html>
<body>
  <h1><% out.println(request.getParameter("titolo")); %></h1>
  <p><% out.println(request.getParameter("testo")); %></p>
</body>
</html>
```

Il ciclo di vita di una JSP è il seguente:

1. la prima volta che viene richiesto un file JSP, Jasper (il motore di Apache TomEE che si occupa della gestione delle JSP) procede alla compilazione del file. La compilazione di tale file genera una servlet che viene caricata in memoria e, dalla successiva richiesta, si procede al suo utilizzo;
2. l'elaborazione della servlet genera l'output HTML, che viene poi inviato al browser;
3. ad una nuova richiesta della stessa JSP, il server verifica se il file.jsp è stato modificato. Se non ci sono state modifiche, viene visualizzata la servlet presente in memoria, altrimenti si procede come per il punto 1.

SCRIPTLET

Come già anticipato, la JSP è un misto tra HTML e codice JAVA.

Uno scriptlet è la porzione di codice JAVA scritta all'interno dei simboli <% %> nelle pagine JSP.

```
<%
if(request.getAttribute("myVar") != null) {
  out.println("my Var non è null!");
} else {
  out.println("my Var è null!");
}
%>
```

DICHIARAZIONI

Le dichiarazioni servono per dichiarare una variabile o un metodo all'interno di una JSP e la sintassi è la seguente:

```
<%! dichiarazione %>
```

Sono sconsigliate le dichiarazioni di metodi nelle JSP, in quanto generalmente è preferibile farle a livello di classe. Questo perché, se inseriamo troppa logica nelle JSP, rischiamo di mischiare troppa logica con il codice HTML, rischiando di creare troppa confusione.

```
<% ! String myVar = request.getParameter("myVar") %>  
  
<% ! public String isPalindroma(String text){  
    ...  
    return ...;  
}  
%>
```

ESPRESSIONI

Un'espressione è una riga di codice che scrive un output all'interno della JSP e la sintassi è la seguente:

```
<%= espressione %>
```

Le espressioni nelle JSP sono molto utilizzate per scrivere i parametri, generati dinamicamente dalle servlet, all'interno di tabelle, o di liste.

```
<% ! String myVar = request.getParameter("myVar") %>  
  
<% ! public String isPalindroma(String text){  
    ...  
    return ...;  
}  
%>  
  
...  
<p>La parola è palindroma? <b><%=isPalindroma(myVar) %></b></p>
```

Altro esempio:

```
<%=request.getParameter("test") != null ? "Test c'è" : "Test non c'è" %>
```

DIRETTIVE

Le direttive sono delle istruzioni particolari che consentono di specificare alcune caratteristiche che sono necessarie per il corretto funzionamento della JSP. Ad esempio, se all'interno di uno scriptlet dobbiamo iterare una lista, quindi dobbiamo fare uso dell'interfaccia List, dobbiamo dire attraverso una direttiva che stiamo utilizzando l'interfaccia java.util.List. Se non scriviamo questa direttiva, avremo un errore in fase di runtime.

Le direttive sono:

- <%@page: definisce alcune impostazioni relative alla compilazione della pagina JSP, attraverso i suoi attributi. I principali attributi sono:
 - language, specifica il linguaggio da utilizzare in fase di compilazione;
Esempio: <%@ page language="Java" %>
 - import, specifica i package da includere nella pagina JSP per il corretto funzionamento;
Esempio: <%@ page import="java.util.List" %>
 - isThreadSafe, se impostato a true, indica che la pagina è in grado di servire più richieste contemporaneamente (cioè viene creato un oggetto per ogni richiesta);
Esempio: <%@ page isThreadSafe="true"%>
- <%@ include: consente di includere all'interno di una JSP altri file (JSP, HTML o file di testo). Questa direttiva consente di creare porzioni di codice HTML e di includerle all'interno di più pagine;
Esempio: <%@ include file="fileB.jsp" %>
- <%@ taglib: consente di utilizzare all'interno della JSP dei custom tag;
Esempio: <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

Vediamo un esempio con alcune direttive:

header.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset-ISO-8859-1">
        <title>Include</title>
    </head>
<body>
    <header>
        <ul>
            <li><a href="">Menu 1</a>
            <li><a href="">Menu 2</a>
        </ul>
    </header>
```

include.jsp

```
<%@ include file = "header.jsp" %>
<% //http://localhost:8080/CORSO_JAVA_EE/jsp/include/include.jsp %>

<section>
    <h1>Sono una JSP!</h1>
</section>

<%@ include file = "footer.jsp" %>
```

footer.jsp

```
<footer>
    Copyright!
</footer>
</body>
</html>
```

Il risultato di questo esempio è il seguente:



AZIONI

Le azioni sono dei componenti (sono delle taglib) messe a disposizione per incapsulare al meglio il codice JAVA presente all'interno di una JSP. Per alcune cose, le azioni sono un'alternativa allo scriptlet.

Le azioni principali sono:

- <jsp:useBean> : permette di associare un'istanza di una classe JavaBean ad una variabile a livello di request (un attributo). Un JavaBean è una classe JAVA che ha le seguenti caratteristiche:
 - ha un costruttore senza argomenti;
 - tutte le sue proprietà sono accessibili usando get, set e is (per le variabili booleane);
 - la classe deve implementare l'interfaccia Serializable;

```
<jsp:useBean id="myvar" class="it.test.Persona" scope="request" />
```

è l'analogo di

```
Persona p = (Persona) request.getAttribute("myvar");
if(p == null) {
    p = new Persona();
    request.setAttribute("myvar", p);
}
```

myvar è una variabile utilizzabile all'interno della jsp.

Vediamo un altro esempio completo con <jsp:useBean> :

ArticoloBeans.java

```
import java.io.Serializable;

/**
 * Classe JavaBean
 */
public class ArticoloBeans implements Serializable {

    private static final long serialVersionUID = -7808622489916673471L;

    private String nome;
    private String codice;
    private double prezzo;

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getCodice() {
        return codice;
    }
}
```

```

public void setCodice(String codice) {
    this.codice = codice;
}

public double getPrezzo() {
    return prezzo;
}

public void setPrezzo(double prezzo) {
    this.prezzo = prezzo;
}
}

```

header.jsp

```

<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
        <title>Beans</title>
    </head>
<body>

```

include.jsp

```

<%@ include file = "header.jsp" %>

<section>
    <h1>Sono una JSP!</h1>

    <jsp:useBean id="articoloSelezionato" class="jsp.beans.ArticoloBeans"
scope="request" />

    <p>
        <% //http:localhost:8080/CORSO_JAVA_EE/jsp/beans/include.jsp
            articoloSelezionato.setNome("giocattolo");
            out.println(articoloSelezionato.getNome());
        %>
    </p>
</section>

<%@ include file = "footer.jsp" %>

```

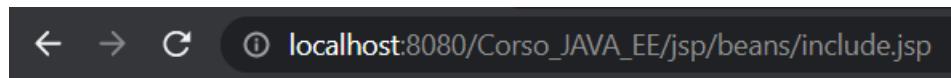
footer.jsp

```

</body>
</html>

```

Il risultato di questo esempio è il seguente:



Sono una JSP!

giocattolo

- <jsp:setProperty> : consente di assegnare un valore ad una variabile di una classe JavaBean;

```
<jsp:setProperty name="myvar" property="username" value="paolo.preite"/>
```

è l'analogo di

```
Persona p = (Persona) request.getAttribute("myvar");
p.setUsername("paolo.preite");
```

- <jsp:getProperty> : consente di recuperare il valore di una variabile di una classe JavaBean;

```
<jsp:getProperty name="myvar" property="username"/>
```

è l'analogo di

```
Persona p = (Persona) request.getAttribute("myvar");
p.getUsername();
```

Vediamo un altro esempio completo con <jsp:setProperty> e <jsp:getProperty> :

ArticoloBeans.java

```
import java.io.Serializable;

/**
 * Classe JavaBean
 */
public class ArticoloBeans implements Serializable {

    private static final long serialVersionUID = -7808622489916673471L;

    private String nome;
    private String codice;
    private double prezzo;

    public String getNome() {
        return nome;
    }
}
```

```

public void setNome(String nome) {
    this.nome = nome;
}

public String getCodice() {
    return codice;
}

public void setCodice(String codice) {
    this.codice = codice;
}

public double getPrezzo() {
    return prezzo;
}

public void setPrezzo(double prezzo) {
    this.prezzo = prezzo;
}
}

```

header.jsp

```

<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
        <title>Beans</title>
    </head>
<body>

```

include.jsp

```

<%@ include file = "header.jsp" %>

<section>
    <h1>Sono una JSP!</h1>

    <jsp:useBean id="articoloSelezionato" class="jsp.beans.ArticoloBeans" scope="request" />

    <p>
        <% //http:localhost:8080/CORSO_JAVA_EE/jsp/beans/include.jsp

            out.println("Articolo " + "<br>");
            articoloSelezionato.setNome("giocattolo");
            articoloSelezionato.setCodice("1111111111");
            articoloSelezionato.setPrezzo(10);
            out.println(articoloSelezionato.getNome() + "<br>");
            out.println(articoloSelezionato.getCodice() + "<br>");
            out.println(articoloSelezionato.getPrezzo() + "<br>");

            // Il blocco di codice sopra è equivalente quello di sotto

            out.println("<br>");
            out.println("Articolo " + "<br>");

        %>

```

```

<jsp:setProperty name="articoloSelezionato" property="nome" value="giocattolo" />
<jsp:setProperty name="articoloSelezionato" property="codice" value="1111111111" />
<jsp:setProperty name="articoloSelezionato" property="prezzo" value="10" />
<jsp:getProperty name="articoloSelezionato" property="nome" />
<br>
<jsp:getProperty name="articoloSelezionato" property="codice" />
<br>
<jsp:getProperty name="articoloSelezionato" property="prezzo" />

</p>
</section>

<%@ include file = "footer.jsp" %>

footer.jsp

</body>
</html>

```

Il risultato di questo esempio è il seguente:



- <jsp:param> : consente di dichiarare ed inizializzare variabili in una JSP. Quest'azione può essere utilizzata anche all'interno delle azioni jsp:include e jsp:forward ;

```

<jsp:params>
    <jsp:param name="nomeParametro" value="valore">
</jsp:params>

```

- <jsp:include> : consente di includere una JSP in un'altra JSP;

```

<jsp:include page="mypage.jsp">
    <jsp:param name="myvar" value="valoremypar">
</jsp:include>

```

La variabile myvar sarà visibile all'interno della jsp mypage.jsp...

- <jsp:forward> : consente di effettuare il forward della richiesta ad un'altra risorsa (html statica, servlet o jsp), interrompendo il flusso in uscita. È sconsigliato eseguire il forward sulle JSP, ma è sempre meglio invece farlo sulle classi;

```
<jsp:forward page="mypage.jsp">
    <jsp:param name="myvar" value="valoremyvar" />
</jsp:forward>
```

La variabile myvar sarà visibile all'interno della jsp mypage.jsp...

OGGETTI IMPLICITI: REQUEST, RESPONSE, OUT, SESSION, APPLICATION

In una JSP possiamo utilizzare i seguenti oggetti impliciti:

- l'oggetto request è di tipo HttpServletRequest e consente di accedere alle informazioni presenti nella richiesta HTTP. Alcuni metodi utilizzati nelle JSP sono:
 - request.getParameter(String param)
 - request.getAttribute(String param)
 - request.getSession().getAttribute(String param)
 - request.setAttribute(String param, Object obj)
 - request.getSession().setAttribute(String param, Object obj)
 - request.getCookies()
 - ...
- l'oggetto response è di tipo HttpServletResponse e consente di gestire i contenuti da inviare al client;
- l'oggetto out è di tipo JspWriter e consente, attraverso i metodi print(Object o) e println(Object o) di stampare del testo (semplice o html) nella JSP.

Esempio

```
<%
    out.print("testo di esempio");
    out.println("testo di esempio"); /* a differenza di print, questo metodo stampa il testo e va a capo.*/
    out.print("<p>Testo paragrafo</p>");
%>
```

- L'oggetto session è di tipo HttpSession e contiene tutte le informazioni relative alla sessione del client. Alcuni metodi utilizzati nelle jsp sono:
 - `session.getAttribute(String param)`
 - `session.setAttribute(String param, Object obj)`
- Alla sessione è possibile accedere anche attraverso l'oggetto request:
- `request.getSession().getAttribute(String param)`
 - `request.getSession().setAttribute(String param, Object obj)`
- L'oggetto application è di tipo ServletContext e consente di accedere alle variabili disponibili a livello di applicazione (quindi variabili indipendenti dall'i-esima request e dalla sessione del client). Tali variabili sono accessibili in ogni pagina o servlet finché l'applicazione è attiva e ha i seguenti metodi:
 - `application.setAttribute(String param, Object obj)` : consente di impostare una variabile a livello di applicazione;
 - `application.getAttribute(String param)`: consente di accedere ad una variabile impostata a livello di applicazione;
 - `application.getRealPath(String path)`: restituisce il percorso assoluto del path specificato (ad esempio `application.getRealPath("test.jsp")` potrebbe restituire `C:/tomcat/webapps/myapp/pages/test.jsp`).

TAG LIBRARY

Come abbiamo visto più volte, per sviluppare un'applicazione web, uno dei principi da utilizzare è il disaccoppiamento tra la parte delle view (o presentation), la parte logica e la parte data.

In particolare, nella parte presentation abbiamo la possibilità di far fronte a questo principio grazie all'utilizzo delle tag library.

Le tag library sono delle componenti XML che possiamo usare nella parte presentation (ossia nelle JSP), che implementano una serie di funzionalità. Per esempio, esistono delle tag library che formattano il testo, altre che vanno in sostituzione dello scriptlet per eseguire un ciclo for o uno statement if e così via.

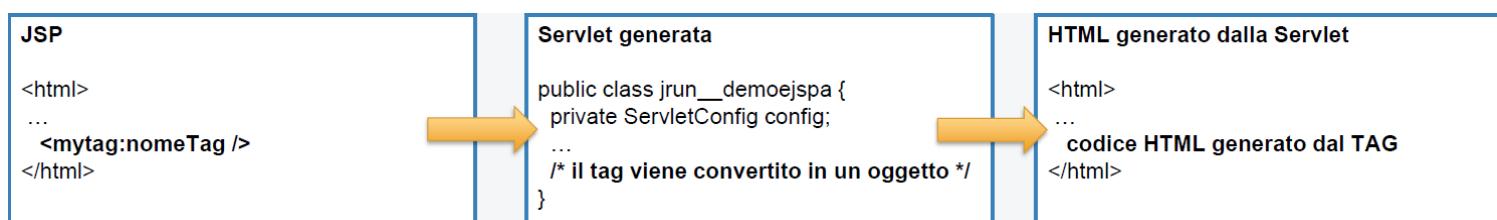
I principali vantaggi nell'usare le tag library sono:

- riutilizzo all'interno di più pagine web ed anche di più web application;
- utilizzo delle tag library al posto di puro codice JAVA nelle scriptlet;
- gestione di logiche applicative riutilizzabili;
- eleganza nella produzione del codice;
- semplicità d'uso anche da parte di non sviluppatori.

Gli elementi che compongono una tag library sono:

- descrittore del tag, che è un file .tld (Tag Library Description) che contiene un documento XML che definisce le caratteristiche del tag;
- una o più classi JAVA, chiamate tag handlers, che definiscono ed implementano la logica. Le classi devono estendere la classe TagSupport, o la classe BodyTagSupport, o devono implementare l'interfaccia Tag. Queste due classi e l'interfaccia si trovano nel package javax.servlet.jsp.tagext.

Alcuni tag generano un output sotto forma di testo semplice, testo formattato o codice HTML. Altri tag implementano logiche applicative.



Abbiamo due tipi di tag:

- Tag senza corpo, che non possono contenere tra i tag di apertura e chiusura altri tag
Esempio: <jsp:useBean id="nomevariabile" class="it.test.ClasseBean" scope="request" />
- Tag con corpo, che possono contenere tra i tag di apertura e chiusura altri tag
Esempio:
<jsp:forward page="nuovaUrl">
 <jsp:param name="nome" value="valore"/>
</jsp:forward>

TAG SENZA CORPO

Un tag senza corpo è una classe JAVA che estende la classe TagSupport.

Alcuni metodi messi a disposizione dalla classe TagSupport, che possiamo sovrascrivere, sono:

- int doEndTag(), viene eseguito alla fine di tutti gli altri metodi e ritorna il valore EVAL_PAGE;
- int doStartTag(), è il metodo principale, quindi eseguito prima di tutti gli altri metodi, e ritorna il valore SKIP_BODY. All'interno di questo metodo implementeremo la logica applicativa del tag. Ad esempio, prendendo il caso di una formattazione di un numero in valuta, all'interno del metodo doStartTag() metteremo la logica che prenderà dalla request il numero che abbiamo passato e lo convertirà in valuta.
- Object getValue(String k), recupera il valore di un attributo definito nel tag.

TAG CON CORPO

Un tag con corpo è una classe JAVA che estende la classe BodyTagSupport (che estende a sua volta la classe TagSupport).

Oltre ad avere i metodi della classe TagSupport visti precedentemente, abbiamo altri metodi messi a disposizione dalla classe BodyTagSupport, che possiamo sovrascrivere, e sono:

- void doInitBody(), invocato solamente una volta, prima della valutazione del corpo del tag. Il metodo è usato per l'inizializzazione di variabili;
- int doAfterBody(), invocato dopo che il corpo del tag è stato valutato. Se questo metodo ritorna EVAL_BODY_TAG, il corpo viene nuovamente valutato, altrimenti se ritorna SKIP_BODY, non viene effettuata la nuova valutazione del corpo del tag.

FILE TDL (TAG LIBRARY DESCRIPTOR)

Il file TLD è un file XML che contiene la descrizione di utilizzo della tag library.

Il file .tdl deve essere inserito nella cartella WEB-INF della web application (generalmente si usa una sottocartella tlds che conterrà tutti i file .tld).

I principali elementi del file XML sono:

- <taglib> : è il nodo root del documento XML che conterrà tutti gli altri nodi;
- <tlib-version> : definisce la versione della taglib utilizzata;
- <jsp-version> : definisce la versione delle specifiche jsp utilizzate;
- <short-name> : contiene un nome descrittivo della tag library;
- <tag> : definisce un tag e contiene al suo interno:
 - <name> : è il nome del tag che utilizzeremo nella JSP;

- <tag-class> : è la classe (completa di package) che implementa il tag;
- <attribute> : definisce un attributo associato al tag e contiene al suo interno:
 - <name> : specifica il nome dell'attributo. Dal punto di vista della classe JAVA, è necessario implementare un metodo set con il nome indicato in questo elemento;
 - <required> : specifica se l'attributo è obbligatorio;
 - <rteprvalue> : specifica se il valore dell'attributo è statico o dinamico, cioè generato utilizzando un'espressione;
 - per i tag con corpo, è necessario specificare anche il seguente elemento: <body-content>JSP</body-content> : in questo modo il contenuto presente tra i tag di apertura e chiusura sarà utilizzabile nella classe JAVA che implementa il tag.

UTILIZZO DELLE TAG LIBRARY NELLE JSP

Per usare una taglib nella JSP si utilizza la seguente direttiva:

```
<%@ taglib uri="/WEB-INF/tld/nomefile.tld" prefix="mytag" %>
```

A questo punto possiamo utilizzare il tag:

```
<mytag:nomeTag nomeAttributo="valore" />
```

JSTL – JSP STANDARD TAG LIBRARY

Le JSTL sono un insieme di tag JSP che implementano le principali funzionalità che utilizziamo comunemente quando sviluppiamo applicazioni web, ad esempio:

- manipolazione di oggetti e stringhe;
- iterazione di liste;
- visualizzazione di parti di HTML in base a determinate condizioni;
- internazionalizzazione dei contenuti;
- ...

Le JSTL sono raggruppate in 5 categorie:

1. core;
2. formattazione;
3. SQL;
4. XML;
5. funzione;

Sono sconsigliati l'utilizzo delle JSTL di tipo SQL e XML, perché hanno interazioni con la parte data

dell'applicazione web, ed è sconsigliato avere questo tipo di interazione direttamente dalle JSP.

Se le JSTL non fanno al caso nostro, possiamo crearcici le nostre tag library personalizzate (vedremo in seguito come fare).

CONFIGURAZIONE DELLA WEB APP PER L'UTILIZZO DI JSTL

Prima di tutto è necessario scaricare il jar contenente tutto il pacchetto JSTL.

Nelle pagine JSP, se vogliamo utilizzare i tag JSTL, dobbiamo inserire la direttiva taglib indicando:

- l'uri, che servirà al container per recuperare dalla mappa tutti i tag handler;
- prefix, che sarà utilizzato nella JSP per scrivere i tag.

Ad esempio, se vogliamo utilizzare nella JSP i tag core, dobbiamo scrivere:

```
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
```

e poi potremo utilizzare i tag nel seguente modo:

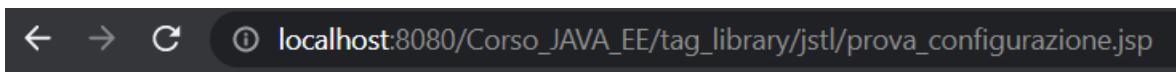
```
<c:if test="">...</c:if>
```

Vediamo un esempio:

prova_configurazione.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
        <title>Prima JSTL</title>
    </head>
    <body>
        <% //http://localhost:8080/CORSO_JAVA_EE/tag_library/jstl/prova_configurazione.jsp %>
        <c:out value="PROVA DI CONFIGURAZIONE!!!"></c:out>
    </body>
</html>
```

Il risultato dell'esempio è il seguente:



PROVA DI CONFIGURAZIONE!!!

TAG CORE

I tag core consentono di effettuare operazioni di iterazione, verifiche condizionali, impostazione e rimozione di variabili, stampa di output e così via.

Per utilizzare i tag core, nelle JSP dobbiamo inserire la seguente direttiva:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
```

Il prefix può essere scelto a piacere, ma per convenzione per i tag core viene sempre utilizzata la lettera “c”.

Vediamo quali sono i principali tag core:

- **<c:out>** : stampa il risultato di un’espressione, oppure una stringa (è analogo di `<%=...%>`), e la sintassi è la seguente:

```
<c:out value="stringa o espressione" default="valore se value è null" escapeXml="
```

Il campo escapeXml, se true, converte i caratteri speciali (esempio `< > & ‘ ’`) nella relativa codifica (esempio `< diventa < ;`) ;

- **<c:set>** : utilizzato per impostare delle variabili con relativo valore e la sintassi è la seguente:

```
<c:set var="nome della variabile" value="valore della variabile" scope="request, session o application"/>
```

Il campo scope indica il livello applicativo in cui andiamo a settare la variabile, che può essere a livello di request, di session o application;

- **<c:remove>** : rimuove una variabile dallo scope impostato e la sintassi è la seguente:

```
<c:remove var="nome della variabile" scope="request, session o application"/>
```

Il campo scope indica il livello applicativo in cui andiamo a rimuovere la variabile, che può essere a livello di request, di session o application;

- **<c:if>** : è analogo dello statement if (senza else). Se l’espressione restituisce true, il tag esegue il codice contenuto nel suo body e la sintassi è la seguente:

```
<c:if test="espressione"  
      var="nome della variabile dov'è salvato il risultato del test"  
      scope="dove sarà salvata la variabile, request, session o application">  
  ...  
</c:if>
```

- **<c:choose>, <c:when>, <c:otherwise>** : sono l’analogo dello statement switch/case e la sintassi è la seguente:

```
<c:choose>  
  <c:when test="espressione da verificare"/>  
  ...  
  </c:when>  
  <c:when test="espressione da verificare"/>  
  ...  
  </c:when>  
  ...  
  <c:otherwise>  
  ...  
  </c:otherwise>  
<c:choose>
```

- <c:import> : è l'analogo di <jsp:include>, solo che consente anche di inserire URL assolute (per esempio <http://www.miosito.it>) e la sintassi è la seguente:

```
<c:import
    url="url path"
    var="variabile dove verrà salvata la url indicata"
    scope="scope della variabile, request, session o application"
    context="opzionale ... simbolo / seguito dal nome di una web application locale" />
```

- <c:forEach> : itera liste di oggetti e la sintassi è la seguente:

```
<c:forEach
    items="lista di oggetti"
    begin=""
    end=""
    step="avanzamento della lista, default è 1"
    var="nome della variabile dove viene salvato l'i-esimo oggetto della lista"
    varStatus="variabile contenente la posizione in cui si trova l'iterazione">
    ...
</c:forEach>
```

- <c:forTokens> : consente di iterare una stringa, specificando il separatore (convertendo, quindi, la stringa in array di stringhe) e la sintassi è la seguente:

```
<c:forTokens
    items="stringa da iterare"
    delims="separatore"
    begin=""
    end=""
    step="avanzamento della lista, default è 1"
    var="nome della variabile dove viene salvato l'i-esimo oggetto della lista"
    varStatus="variabile contenente la posizione in cui si trova l'iterazione">
    ...
</c:forTokens>
```

- <c:url> : formatta una stringa in URL e la sintassi è la seguente:

```
<c:url
    var="nome della variabile che conterrà la URL generata"
    scope="scope della variabile"
    value="stringa da convertire in URL"
    context="opzionale ... simbolo / seguito dal nome di una web application locale">
    ...
</c:url>
```

- <c:param> : utilizzato nel <c:url> per specificare eventuali parametri e la sintassi è la seguente:

```
<c:param name="nome della variabile" value="valore salvato nella variabile"/>
```

Esempio

```
<c:url var="nomeVar" value="test.jsp">
    <c:param name="method" value="demo"/>
</c:url>
```

Vediamo un esempio di utilizzo dei tag core:

core.jsp

```
<%@ page import="java.util.* , java.io.*" %>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset-ISO-8859-1">
        <title>Prima JSTL</title>
    </head>
<body>
    <% //http://localhost:8080/CORSO_JAVA_EE/tag_library/jstl/core.jsp %>

    <c:if test="${param.test1 == null && param.test2 == null}">
        I parametri test1 e test2 non hanno alcun valore nella request
    </c:if><br>
    <c:if test="${param.test1 == null}">
        Il parametro test1 non ha alcun valore nella request
    </c:if><br>
    <c:if test="${param.test2 == null}">
        Il parametro test2 non ha alcun valore nella request
    </c:if><br>

    <% //Le due righe di stampa seguenti sono equivalenti: %> <br>
    <c:out value="<%=>request.getParameter("test1") %>" default="ECCO!"></c:out> <br>
    <c:out value="${param.test2}" default="RIECCO!"></c:out> <br>

    <c:set var="prova" value="1234" scope="request"></c:set>
    <c:out value="${prova}" default="RIECCOLO!"></c:out> <br>
    <% /*È possibile stampare il valore di una variabile
        anche direttamente tra tag HTML, nel seguente modo: */%>
    <h2>${prova}</h2>

    <c:choose>
        <c:when test="${param.test1 == null && param.test2 == null}">
            I parametri test1 e test2 non hanno alcun valore nella request (sono nello switch case)
        </c:when>
        <c:when test="${param.test1 == null}">
            Il parametro test1 non ha alcun valore nella request (sono nello switch case)
        </c:when>
        <c:when test="${param.test2 == null}">
            Il parametro test2 non ha alcun valore nella request (sono nello switch case)
        </c:when>
        <c:otherwise>
            Nessuna condizione dello switch case si e' verificata!
        </c:otherwise>
    </c:choose> <br>
    <br>

    <%
        List<String> lista = new ArrayList<String>();
        lista.add("Valore 1");
        lista.add("Valore 2");
        lista.add("Valore 3");
        lista.add("Valore 4");
        lista.add("Valore 5");

        request.setAttribute("elementi", lista);
    /*
        requestScope.NOMEATTRIBUTO è l'equivalente di request.getAttribute("NOMEATTRIBUTO")
        sessionScope.NOMEATTRIBUTO è l'equivalente di session.getAttribute("NOMEATTRIBUTO")
    
```

```

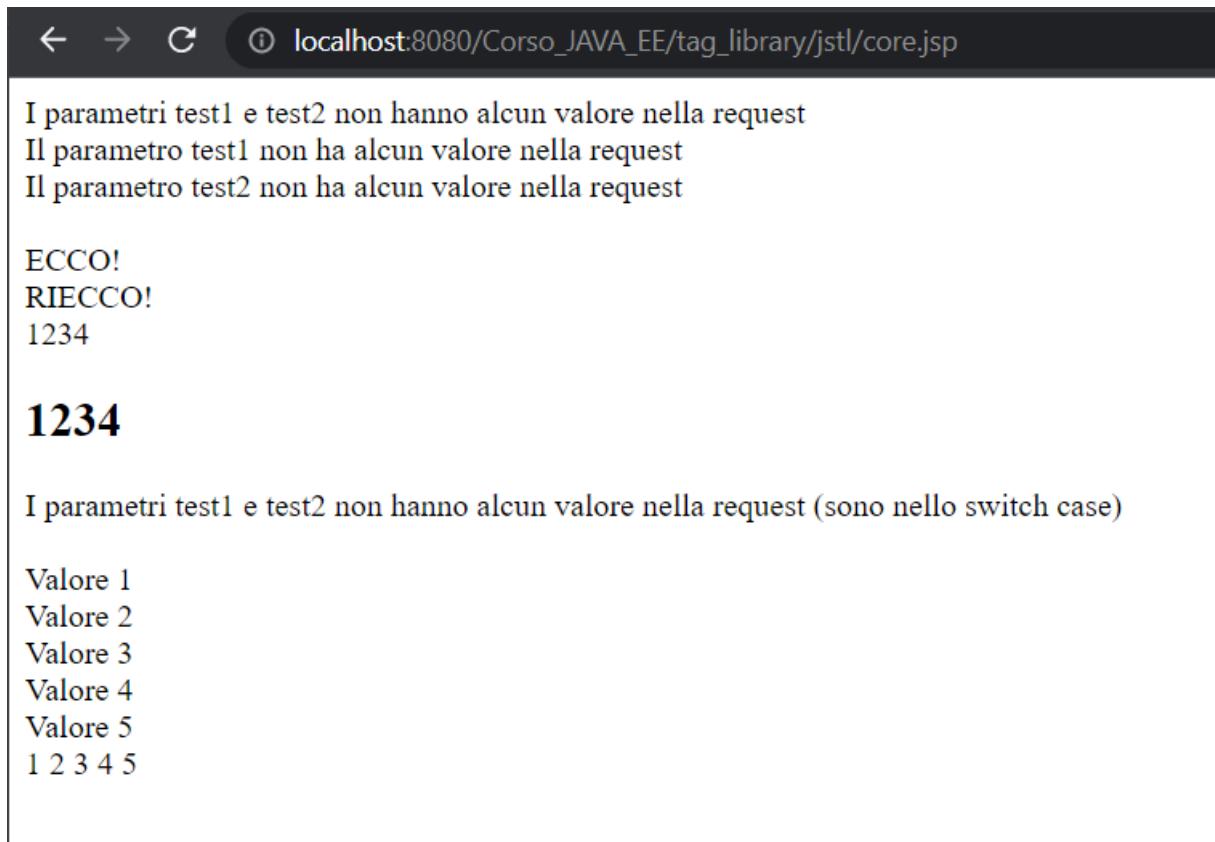
        param.NOMEPARAMETRO è l'equivalente di request.getParameter("NOMEATTRIBUTO")
    */
%>
<c:forEach items="${requestScope.elementi}" var="elemento" varStatus="contatore">
    ${elemento}<br>
</c:forEach>

<c:forTokens items="1,2,3,4,5" delims="," var="numero" varStatus="contatore">
    ${numero}
</c:forTokens>

</body>
</html>

```

I risultati dell'esempio, in base ai parametri inseriti nell'URL, sono i seguenti:



The screenshot shows a browser window with the URL `localhost:8080/CORSO_JAVA_EE/tag_library/jstl/core.jsp`. The page content is as follows:

```

I parametri test1 e test2 non hanno alcun valore nella request
Il parametro test1 non ha alcun valore nella request
Il parametro test2 non ha alcun valore nella request

ECCO!
RIECCO!
1234

1234

I parametri test1 e test2 non hanno alcun valore nella request (sono nello switch case)

Valore 1
Valore 2
Valore 3
Valore 4
Valore 5
1 2 3 4 5

```

Il parametro test2 non ha alcun valore nella request

prova1
RIECCO!
1234

1234

Il parametro test2 non ha alcun valore nella request (sono nello switch case)

Valore 1
Valore 2
Valore 3
Valore 4
Valore 5
1 2 3 4 5

Il parametro test1 non ha alcun valore nella request

ECCO!
prova2
1234

1234

Il parametro test1 non ha alcun valore nella request (sono nello switch case)

Valore 1
Valore 2
Valore 3
Valore 4
Valore 5
1 2 3 4 5

prova1
prova2
1234

1234

Nessuna condizione dello switch case si e' verificata!

Valore 1
Valore 2
Valore 3
Valore 4
Valore 5
1 2 3 4 5

TAG FORMAT

I tag format consentono, come dice il nome, di effettuare operazioni di formattazione di testo, numeri, date, stringhe e così via.

Per utilizzare i tag format, nelle JSP dobbiamo inserire la seguente direttiva:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fnt"%>
```

Vediamo quali sono i principali tag format:

- <fmt:formatNumber> : consente di formattare i numeri (in valuta, percentuali e così via) e la sintassi è la seguente:

```
<fmt:formatNumber  
    value="valore numerico da formattare"  
    type="tipo di numero da generare: NUMBER, CURRENCY, o PERCENT"  
    pattern="formato del numero stampato (ad esempio ###.##)"  
    currencyCode="codice della valuta da utilizzare per formattare il valore"  
    currencySymbol="simbolo della valuta da utilizzare per formattare il valore"  
    maxIntegerDigits="numero massimo di interi da stampare"  
    minIntegerDigits="numero minimo di interi da stampare"  
    maxFractionDigits="numero massimo di cifre decimali da stampare"  
    minFractionDigits="numero minimo di cifre decimali da stampare"  
    var="variabile dove salvare il valore formattato"  
    scope="scope della variabile" >  
</fmt:formatNumber>
```

Alcuni valori da utilizzare per definire il pattern

0	Rappresenta una cifra
E	Rappresenta il formato esponenziale
#	Rappresenta una cifra. Visualizza 0 se non ci sono cifre disponibili
.	Separatore dei decimali
,	Separatori delle migliaia
-	Utilizzare il prefisso negativo di default
%	Visualizza il numero a multipli di 100 e in formato percentuale
?	Visualizza il numero a multipli di 1000 e in formato per mille
¤	Rappresenta il simbolo della valuta. Questo elemento viene sostituito con la valut a correntemente impostata

- <fmt:parseNumber> : utilizzato per convertire stringhe (che possono essere numeri, percentuali, valuta) in numeri e la sintassi è la seguente:

```
<fmt:parseNumber  
    value="stringa contenente il numero da elaborare"  
    type="tipo di numero da generare: NUMBER, CURRENCY o PERCENT"  
    pattern="formato del numero stampato (as esempio ###.##)"  
    parseLocale="Locale da utilizzare quando si fa il parsing della stringa"  
    integerOnly="se true, indica che la stringa verrà convertita in intero"  
    var="variabile dove salvare il valore formattato"  
    scope="scope della variabile">  
</fmt:parseNumber>
```

- <fmt:formatDate> : utilizzato per formattare le date e la sintassi è la seguente:

```
<fmt:formatDate
    value="oggetto Date da formattare"
    type="tipo di output atteso: DATE, TIME or BOTH"
    dateStyle="FULL, LONG, MEDIUM, SHORT or DEFAULT"
    timeStyle="FULL, LONG, MEDIUM, SHORT or DEFAULT"
    pattern="formato personalizzato di visualizzazione della data"
    var="variabile dove salvare il valore formattato"
    scope="scope della variabile"/>
```

Alcuni valori da utilizzare per definire il pattern

y	Anno	2002
M	Mese	Aprile oppure 04
d	Giorno del mese	20
h	Ora del giorno (12-hour time)	12
H	Ora del giorno (24-hour time)	0
m	Minuti	45
s	Secondi	52
S	Millisecondi	970
E	Giorno della settimana	Martedì
D	Giorno dell'anno	180

Vediamo un esempio di utilizzo dei tag format:

format.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"%>
<%@ page import="java.util.* , java.io.*" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
        <title>Format</title>
    </head>
<body>
    <% //http://localhost:8080/CORSO_JAVA_EE/tag_library/jstl/format.jsp %>

    <fmt:formatNumber
        var="numero"
        value="1000.45677"
        type="CURRENCY"
        maxFractionDigits="2"
        minFractionDigits="2"
        pattern="###.##">
    </fmt:formatNumber>
    <p> Valore formattato ${numero}</p>

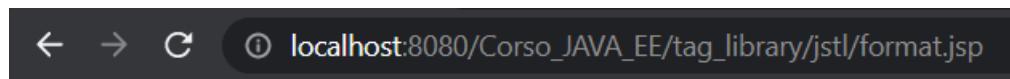
    <%
        Calendar c = new GregorianCalendar();
    %>
```

```
c.set(Calendar.YEAR, 2017);
c.set(Calendar.MONTH, 9);
c.set(Calendar.DATE, 28);

    request.setAttribute("dataCorrente", c.getTime());
%>
<fmt:formatDate
    value="${requestScope.dataCorrente}"
    pattern="d/M/y"
    var="data"/>
<p> Data formattata ${data}</p>

</body>
</html>
```

L'output di questo esempio è il seguente:



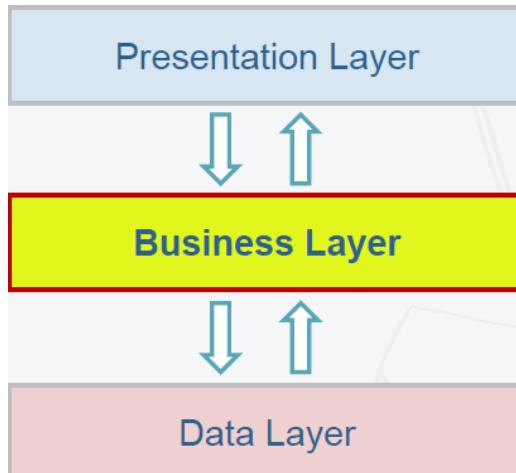
Valore formattato 1000,46

Data formattata 28/10/2017

ENTERPRISE JAVA BEAN (EJB)

Un EJB è un componente software (in poche parole una classe JAVA) che dobbiamo utilizzare quando sviluppiamo applicazioni JAVA EE per implementare la logica di business dell'applicazione.

Per logica di business si intende l'insieme di tutte quelle funzionalità che un'applicazione deve avere per poter eseguire determinate operazioni.



Un EJB, per essere tale, deve rispettare le seguenti caratteristiche:

- persistenza;
- supporto alle transazioni;
- gestione della concorrenza;
- gestione della sicurezza;
- integrazione con altre tecnologie, ad esempio JMS (Java Message Service) per la gestione delle code dei messaggi, JNDI che è un database e CORBA che è uno strumento di integrazione con applicativi sviluppati con altre tecnologie.

Creare un EJB è molto semplice, in quanto possiamo creare una normale classe JAVA e utilizzare le annotation per definire l'EJB.

Un'altra caratteristica importante che offrono gli EJB, potendo utilizzare al loro interno le annotation, è che effettuano la dependency injection.

Per poter far funzionare un EJB, è necessario che l'applicazione venga installata all'interno di un EJB container. Un EJB container è un software che si trova all'interno del Referencing Runtimes. Come abbiamo già visto, un Referencing Runtimes contiene al suo interno una serie di librerie già disponibili, tra cui il servlet container per la gestione del presentation layer (gestione di servlet e JSP).

Abbiamo due tipi di EJB:

- Session Bean (Stateless e Stateful);
- Message Driven Bean;

DEPENDENCY INJECTION

La dependency injection è un pattern di programmazione, così come lo sono il singleton e l'MVC. Questo pattern consente di iniettare delle dipendenze all'interno di un'altra classe.

Vediamo un esempio pratico:

1) Senza dependency injection

Smartphone.java

```
public class Smartphone {  
    private Fotocamera camera;  
  
    public Smartphone() {  
    }  
  
    public void scattaFoto() {  
        camera = new Fotocamera();  
        camera.scatta();  
    }  
}
```

2) Con dependency injection

Smartphone.java

```
public class Smartphone {  
    private Fotocamera camera;  
  
    public Smartphone(Fotocamera camera) {  
        this.camera = camera;  
    }  
}
```

Test.java

```
class Test {  
    public static void main(String[] args) {  
        Fotocamera fc = new Fotocamera;  
        Smartphone s=new Smartphone(fc);  
    }  
}
```

Partiamo con il creare una classe Smartphone. La classe Smartphone ha tra i suoi attributi un oggetto "camera" di tipo Fotocamera. Senza la dependency injection, la creazione dell'istanza di "camera" è in carico alla classe Smartphone. Con la dependency injection, invece, l'oggetto "camera" viene popolato (inizializzato tramite new) per noi da un'altra classe, perché fa parte del costruttore della classe Smartphone.

Quindi si parla di dependency injection quando una classe non si deve occupare di popolare i suoi attributi (dell'inizializzazione dei suoi attributi tramite il new), ma tali attributi verranno passati (iniettati) ad un'altra classe e verranno popolati da quest'ultima (dipendenza da un'altra classe).

Esistono tre tipi di dependency injection:

- Constructor Injection, ossia la variabile viene inizializzata nel costruttore (come nell'esempio precedente);
- Setter Injection, ossia la variabile viene inizializzata nel suo metodo `setNomeVariabile`;
- Interface injection, ossia il metodo `set` si trova nell'interfaccia che è implementata dalla classe che ha la dependency.

EJB STATELESS

Come già visto, un EJB è un componente dello strato business che consente di sviluppare funzionalità che implementano la logica di business di un'applicazione.

Gli EJB stateless sono un tipo di EJB che, tra una richiesta e l'altra, non mantengono informazioni sul client che ha effettuato la richiesta, analogamente al protocollo HTTP. Anche lo stesso termine stateless sta ad indicare che questo EJB non salva le informazioni sulla connessione tra il componente che l'ha chiamato e se stesso.

Gli EJB stateless sono i più utilizzati in assoluto, perché consentono di sviluppare tutte quelle funzionalità classiche di un'applicazione, come ad esempio la creazione delle fatture, le scritture contabili, la generazione di un bilancio, il salvataggio di un ordine e così via.

Vediamo l'esempio di creazione di una EJB stateless (anche in questo caso è richiesto il download della libreria esterna prima di poter utilizzare gli EJB, qualora non sia già stata configurata nel progetto):

```
import javax.ejb.LocalBean;
import javax.ejb.Stateless;

@Stateless
@LocalBean
public class EJBStateless {

    public EJBStateless() {
    }

    public String saluto() {
        return "Ciao sono un EJB!";
    }
}
```

Vediamo anche cosa è JNDI, perché è un componente fondamentale dell'architettura JAVA EE che regola l'accesso ai servizi installati, tra cui gli EJB.

Quando effettuiamo il deploy (l'installazione) di un EJB, o l'installazione di un'applicazione web che al suo interno contiene gli EJB, ognuno di essi deve essere registrato all'interno del JNDI (Java Naming Directory Interface).

Il JNDI è un registro che mappa tutti i nostri servizi installati nel server (ad esempio gli EJB), associandoli ad un nome. Ogni richiesta di servizio (all'interno dello stesso ambiente o su una macchina remota) passa da JNDI.

Quindi ogni EJB, che viene deployata all'interno dell'applicazione, ha un nome all'interno del nostro server. Per accedere a questo nome ci sarà un annotation apposita che richiama l'EJB, effettuando una chiamata alla JNDI.

Quando creiamo un EJB, il client che lo invoca non accede direttamente alla classe dell'EJB stesso, ma ad una sua interfaccia. Questo perché, ricollegandoci al discorso dell'information hiding, l'interfaccia dice cosa fa quel servizio, ma non ti dice internamente come lo fa.

Le interfacce legate all'EJB possono essere di due tipi:

- interfacce local , che possono essere utilizzate da client che si trovano nello stesso Referencing Runtimes in cui è attivo l'EJB;
- interfacce remote, che possono essere utilizzate da client che si trovano su un altro Referencing Runtimes rispetto a dove è attivo l'EJB.

L'interfaccia è una sorta di contratto verso il client, perché contiene la definizione dei metodi che l'EJB è in grado di fare.

Il client che invoca un metodo di un'interfaccia dell'EJB, interagisce con un oggetto noto come EJB Object. L'EJB Object è uno Stub che implementa le interfacce local e remote ed è responsabile dell'inoltro della chiamata verso l'EJB Container.

Per creare EJB stateless dobbiamo utilizzare la seguente annotation:

- javax.ejb.Stateless, che deve essere applicata sulla definizione di una classe.

Per definire le interfacce di un EJB stateless, dobbiamo fare nel seguente modo:

- javax.ejb.Local, per definire un'interfaccia local;
- javax.ejb.Remote, per definire un'interfaccia remota.

Se non specifichiamo un'interfaccia, l'EJB verrà considerato local.

Esempio

```
@Stateless  
public class MioBean implements MioBeanLocal, MioBeanRemote {  
    ...  
}  
  
@Local  
public interface MioBeanLocal {  
    ...  
}  
  
@Remote  
public interface MioBeanRemote {  
    ...  
}
```

Vediamo un esempio di definizione di interfaccia locale e remota per la creazione di un EJB stateless:

EJBStatelessPadre.java

```
public interface EJBStatelessPadre {  
    public String saluto();  
    public void stampaFattura();  
}
```

EJBStatelessLocal.java

```
import javax.ejb.Local;  
  
@Local  
public interface EJBStatelessLocal extends EJBStatelessPadre{  
    public void salvaFattura();  
}
```

EJBStatelessRemote.java

```
import javax.ejb.Remote;  
  
@Remote  
public interface EJBStatelessRemote extends EJBStatelessPadre{  
    public void collegamentoServizio();  
}
```

EJBStateless.java

```
import javax.ejb.LocalBean;  
import javax.ejb.Stateless;  
  
@Stateless  
@LocalBean  
public class EJBStateless implements EJBStatelessLocal, EJBStatelessRemote {  
  
    public EJBStateless() {  
    }  
  
    @Override  
    public String saluto() {  
        return "Ciao sono un EJB!";  
    }  
  
    @Override  
    public void stampaFattura() {  
    }  
  
    @Override  
    public void salvaFattura() {  
    }
```

```

@Override
public void collegamentoServizio() {
}
}

```

Il fatto di creare un'interfaccia padre che estenda l'interfaccia remota e locale serve per definire una soluzione elegante in cui vado a effettuare la dichiarazione dei metodi dell'EJB solo una volta nell'interfaccia padre, senza dover dichiarare più volte lo stesso metodo sia nell'interfaccia locale che in quella remota. Tuttavia, nel caso in cui voglia dichiarare dei metodi solo nell'interfaccia remota o locale, basta dichiarare tale metodo nell'interfaccia desiderata, senza doverlo dichiarare nell'interfaccia padre.

Fino a questo momento abbiamo solamente creato l'EJB, ma ancora non siamo in grado di utilizzarlo.

Per utilizzare un EJB all'interno di un client (ad esempio in una servlet, o in una web application), dobbiamo utilizzare la seguente annotation:

- javax.ejb.EJB, che deve essere applicata sull'istanza dell'interfaccia (locale o remota) definita nel client.

Esempio di accesso ad un'interfaccia Local

```

@EJB
private MioBeanLocal nomeVariabileBean;

```

Ovviamente, se implementiamo un EJB senza interfacce, nel client dobbiamo creare una variabile di tipo classe del bean.

Esempio di accesso ad un'interfaccia Remote

```

@EJB
private MioBeanRemote nomeVariabileBean;

```

Vediamo un esempio di richiamo dell'EJB creato nell'esempio precedente:

EsegueoEJBStatelessServlet.java

```

import javax.ejb.EJB;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@WebServlet("/RichiamoEJBStateless") //http://localhost:8080/CORSO_JAVA_EE/RichiamoEJBStateless
public class EsegueoEJBStatelessServlet extends HttpServlet {

    @EJB
    EJBStatelessLocal ejbStatelessLocal;

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        String saluto = ejbStatelessLocal.saluto();
        req.setAttribute("salutoEJB", saluto);
        req.getServletContext().getRequestDispatcher("/service/EJB_stateless/RichiamoEJBStateless.jsp").include(req,resp);
    }

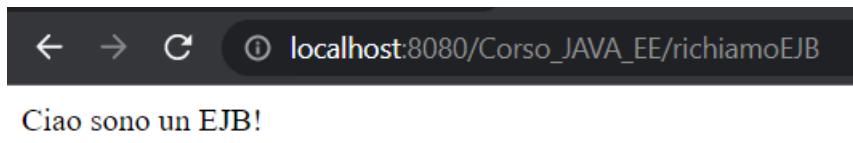
    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
    }
}

```

RichiamoEJBStateless.jsp

```
<%@ page import="java.util.* , java.io.*" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
        <title>Richiamo EJB Stateless</title>
    </head>
    <body>
        <c:set var="salutoArrivatoDaEJB" value="${requestScope.salutoEJB}" scope="request"></c:set>
        <c:out value="${requestScope.salutoArrivatoDaEJB}" default="Non è arrivato nessun saluto!"></c:out> <br>
    </body>
</html>
```

Il risultato di questo esempio è il seguente:



METODI ASINCRONI IN UN EJB

In un EJB è possibile creare anche metodi asincroni. I metodi asincroni sono quelli che non necessitano necessariamente di una risposta immediata.

Per implementare un metodo asincrono, è necessario utilizzare l'annotation `@Asynchronous`.

Il valore che il metodo asincrono deve ritornare (la risposta da inviare al client), deve essere di tipo `java.util.concurrent.Future<V>`, dove V è il tipo del risultato ritornato (ad esempio `Future<Integer>` se il metodo ritorna un intero).

L'oggetto `Future` implementa il metodo `isDone()`, che consente di verificare lo stato dell'esecuzione del metodo asincrono.

Esempio di accesso ad un'interfaccia Local

```
public Future<String> generaBilancio(int anno) {
    ...
}
```

L'oggetto `Future` restituito ha il metodo `isDone()` che consente di verificare lo stato dell'esecuzione del metodo.

GESTIONE DELLE TRANSAZIONI NEGLI EJB CONTAINER-MANAGED

Quando invochiamo più metodi in una EJB, questi di default vengono eseguiti in transazioni tra loro. Questo vuol dire che, se un metodo fallisce e genera un'eccezione, su tutti gli altri verrà effettuato il rollback, quindi tutte le istruzioni che erano state eseguite non avranno più alcun effetto.

Per impostare manualmente il comportamento transazionale di un metodo, dobbiamo inserire sul metodo l'annotation `@TransactionAttribute`, sulla quale dobbiamo impostare il valore del `TransactionAttributeType`, che può essere:

- `NotSupported`: il metodo non supporta le transazioni. Se invocato in una transazione in corso, questa viene sospesa durante la sua esecuzione;
- `Supports`: il metodo supporta le transazioni. Il metodo non apre alcuna transazione, ma utilizza una transazione già attiva;
- `Required` (default): il metodo si unisce ad una transazione in corso. Ne apre una nuova se non esistono transazioni;
- `RequiredNew`: il metodo apre sempre una transazione che si chiude al termine dell'esecuzione del metodo;
- `Mandatory`: il metodo deve sempre essere eseguito all'interno di una transazione;
- `Never`: il metodo non può essere eseguito mai all'interno di una transazione.

EJB STATEFUL

Per creare un EJB stateful bisogna creare una classe e mettere l'annotation `@Stateful`. Per il resto il procedimento di creazione è lo stesso degli EJB stateless, ossia si creano le interfacce locali e remote se vogliamo disporre di entrambe, o solo di una delle due.

Gli EJB stateful consentono di mantenere una sessione attiva con il client, per cui quando invochiamo un metodo dell'EJB stateful, l'EJB mantiene la persistenza delle informazioni, cioè mantiene la sessione attiva tra se stesso e il client.

Per mantenere lo stato dell'EJB stateful, devono essere definite delle variabili all'interno della classe dell'EJB. Tali variabili devono essere tipi primitivi o implementare l'interfaccia `Serializable`.

Vediamo l'esempio di creazione e richiamo di una EJB stateful (anche in questo caso è richiesto il download della libreria esterna prima di poter utilizzare gli EJB, qualora non sia già stata configurata nel progetto):

EJBStatefulLocal.java

```
import javax.ejb.Local;

@Local
public interface EJBStatefulLocal {
    public int incrementa(int c);
}
```

EJBStateful.java

```
import javax.ejb.LocalBean;
import javax.ejb.Stateful;

@Stateful
@LocalBean
public class EJBStateful implements EJBStatefulLocal{
    private int contatore;

    public EJBStateful() {
    }

    public int getContatore() {
        return contatore;
    }

    public void setContatore(int contatore) {
        this.contatore = contatore;
    }

    @Override
    public int incrementa(int c) {
        setContatore(getContatore() + c);
        return getContatore();
    }
}
```

EseguoEJBStatefulServlet.java

```
import javax.ejb.EJB;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@WebServlet("/RichiamoEJBStateful") //http://localhost:8080/CORSO_JAVA_EE/RichiamoEJBStateful
public class EseguoEJBStatefulServlet extends HttpServlet{

    @EJB
    EJBStatefulLocal ejbStatefulLocal;

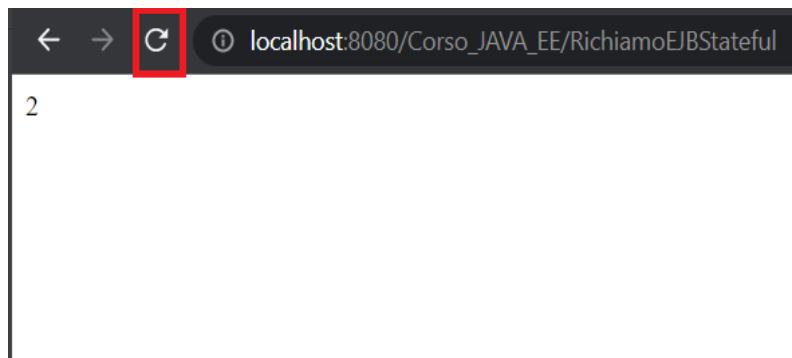
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        req.setAttribute("contatore", ejbStatefulLocal.incrementa(1));
        req.getServletContext().getRequestDispatcher("/service/EJB_stateful/RichiamoEJBStateful.jsp").include(req,resp);
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        super.doPost(req, resp);
    }
}
```

RichiamoEJBStateful.jsp

```
<%@ page import="java.util.* , java.io.*" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
        <title>Richiamo EJB Stateful</title>
    </head>
<body>
    <c:set var="contatoreIncrementatoDaEJB" value="${requestScope.contatore}" scope="request"></c:set>
    <c:out value="${requestScope.contatore}" default="Non è arrivato nessun contatore!"></c:out> <br>
</body>
</html>
```

Il risultato di questo esempio è il seguente:



e così via...

Due funzionalità importanti che hanno gli EJB stateful sono:

- Passivazione: se tra client e l'EJB stateful non ho attività per un certo intervallo di tempo, il server rimuove dalla memoria l'EJB e salva il suo stato sul disco, liberando la RAM per renderla disponibile ad altri processi;
- Attivazione: alla successiva invocazione di un metodo dell'EJB stateful da parte del client, il server procede con la riattivazione, creando un oggetto in memoria che si trova nello stato salvato precedentemente.

Attraverso le annotation `@PrePassivate` e `@PostActivate` possiamo implementare dei metodi per effettuare delle operazioni prima che l'EJB stateful venga salvato sul disco e dopo che l'EJB stateful è stato riattivato.

Generalmente i metodi che utilizzano queste annotation si occupano di rilasciare risorse per la gestione di transazioni, accessi al database e così via.

Se all'interno dell'EJB stateful definiamo un metodo con l'annotation `@Remove`, una volta invocato dal client avremo la chiusura della connessione e la rimozione in memoria dell'EJB.

Vediamo un esempio con l'annotation `@Remove`:

EJBStatefullLocal.java

```
import javax.ejb.Local;

@Local
public interface EJBStatefullLocal {
    public int incrementa(int c);
    public void chiudiConnessione();
}
```

EJBStateful.java

```
import javax.ejb.LocalBean;
import javax.ejb.Remove;
import javax.ejb.Stateful;

@Stateful
@LocalBean
public class EJBStateful implements EJBStatefullLocal{
    private int contatore;

    public EJBStateful() {}

    public int getContatore() {
        return contatore;
    }

    public void setContatore(int contatore) {
        this.contatore = contatore;
    }

    @Override
    public int incrementa(int c) {
```

```

        setContatore(getContatore() + c);
        return getContatore();
    }

    @Override
    @Remove
    public void chiudiConnessione() {

    }
}

```

EseguoEJBStatefulServlet.java

```

import javax.ejb.EJB;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@WebServlet("/RichiamoEJBStateful") //http://localhost:8080/CORSO_JAVA_EE/RichiamoEJBStateful
public class EseguoEJBStatefulServlet extends HttpServlet{

    @EJB
    EJBStatefulLocal ejbStatefulLocal;

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        req.setAttribute("contatore", ejbStatefulLocal.incrementa(1));
        if (req.getAttribute("contatore").equals(2)){
            ejbStatefulLocal.chiudiConnessione();
        }
        req.getServletContext().getRequestDispatcher("/service/EJB_stateful/RichiamoEJBStateful.jsp").include(req,resp);
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        super.doPost(req, resp);
    }
}

```

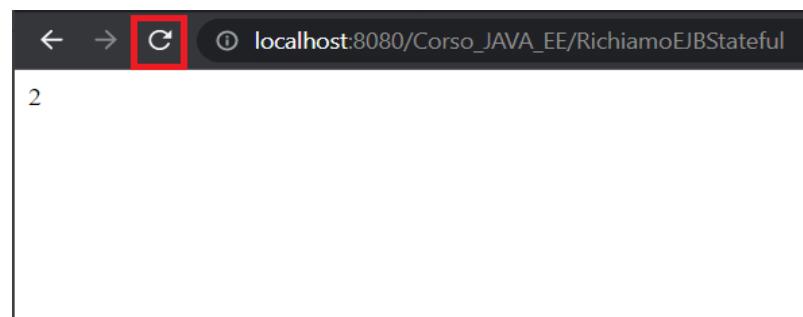
RichiamoEJBStateful.jsp

```

<%@ page import="java.util.* , java.io.*" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
        <title>Richiamo EJB Stateful</title>
    </head>
    <body>
        <c:set var="contatoreIncrementatoDaEJB" value="${requestScope.contatore}" scope="request"></c:set>
        <c:out value="${requestScope.contatore}" default="Non è arrivato nessun contatore!"></c:out> <br>
    </body>
</html>

```

Il risultato di questo esempio è il seguente:

A screenshot of a web browser window showing an error page. The address bar shows the URL "localhost:8080/CORSO_JAVA_EE/RichiamoEJBStateful". The letter 'C' in the address bar is highlighted with a red box.

HTTP Status 500 – Internal Server Error

Type Exception Report

Message Not Found

Description The server encountered an unexpected condition that prevented it from full

Exception

```
javax.ejb.NoSuchEJBException: Not Found
    org.apache.openejb.core.ivm.BaseEjbProxyHandler.convertExc
    org.apache.openejb.core.ivm.BaseEjbProxyHandler.invoke(Bas
com.sun.proxy.$Proxy93.incrementa(Unknown Source)
```

È possibile anche disattivare la passivazione. In questo caso, l'EJB stateful rimane in memoria anche nei periodi di inattività.

La disattivazione della passivazione si ottiene mediante l'uso dell'attributo `passivationCapable = false` dell'annotation `@Stateful`.

QUANDO USARE EJB STATELESS O EJB STATEFUL

Nella maggior parte dei casi si utilizzano gli EJB stateless perché, non mantenendo lo stato, non occupano memoria inutilmente all'interno del server. Immaginando di avere un'applicazione web che si occupa di e-commerce (dove abbiamo l'elenco degli ordini, l'elenco delle fatture, il catalogo dei prodotti, la gestione del magazzino e così via), tutte le attività di accesso alle varie liste o cataloghi per effettuare una ricerca sono attività che devono essere effettuate su richiesta, quindi in quel caso utilizziamo gli EJB stateless.

Gli EJB stateful, invece, vanno utilizzati quando ci troviamo in situazioni in cui dobbiamo implementare delle funzionalità che richiedono di mantenere una connessione attiva tra l'EJB e il client che lo ha invocato.

INTERCEPTOR

Quando abbiamo parlato delle web application, in particolare delle servlet, abbiamo introdotto il concetto di servlet filter. Abbiamo visto che un servlet filter è una componente che intercetta una determinata request su un path specificato.

Analogamente al concetto di servlet filter che riguarda le richieste HTTP, gli interceptor sono delle classi JAVA che contengono dei metodi che vengono eseguiti prima della chiamata ai metodi di un EJB, ponendosi quindi tra il client e l'EJB.

Per creare un interceptor, basta creare una normalissima classe JAVA che ha le seguenti caratteristiche:

- può contenere una serie di metodi;
- uno solo dei metodi deve essere marcato con l'annotation `@AroundInvoke`. Il metodo con questa annotation deve avere la seguente struttura:
 - in input deve avere un oggetto di tipo `InvocationContext` e in output deve restituire un oggetto `Object`;
 - il metodo può avere un accesso `public`, `private`, `protected` e non deve essere `static` o `final`.

```
@AroundInvoke  
public Object mioMetodo(InvocationContext ctx) throws Exception {  
    ...  
}
```

Vediamo un esempio di creazione dell'interceptor:

Interceptor.java

```
import javax.interceptor.AroundInvoke;  
import javax.interceptor.InvocationContext;  
  
public class Interceptor {  
  
    public void log(String dato) {  
        System.out.println(dato);  
    }  
}
```

```

@AroundInvoke
public Object filtra(InvocationContext ic) throws Exception {
    log("FILTRO!");
    System.out.println("sono nel metodo filtra!");

    return "OK";
}

```

I metodi definiti nell’interfaccia InvocationContext sono:

- `public Object getTarget()`: restituisce un riferimento all’istanza dell’EJB intercettato;
- `public Method getMethod()`: consente di recuperare un oggetto di riferimento per il metodo invocato sull’EJB;
- `public Object[] getParameters()` e `public void setParameters(Object[] new Args)`: consentono di operare sui parametri di input del metodo intercettato;
- `public Map<String, Object> getContextData()`: consente di passare dati tra interceptor, nel caso in cui in un EJB abbiamo utilizzato una catena di interceptor;
- `public Object proceed()`: indica che si può andare avanti nella catena degli interceptor;
- `public Object getTimer()`: ritorna un oggetto timer associato all’invocazione di un metodo dell’EJB che gestisce il timeout.

Un interceptor può essere associato:

- all’EJB: in questo caso verrà attivato ogni volta che un metodo dell’EJB viene invocato;
- ad uno o più metodi dell’EJB: in questo caso verrà attivato solo quando il metodo o i metodi annotati verranno invocati.

Per collegare un interceptor all’EJB, o al metodo, dobbiamo annotare la classe, o il metodo, con l’annotation `@Interceptors`, specificando la classe dell’interceptor (o le classi se gli interceptor sono più di uno).

Per associare più interceptor ad un EJB, o ad un metodo, nell’annotation `@Interceptors` possiamo inserire un array di classi: `@Interceptors({PrimoInterceptor.class, SecondoInterceptor.class, ...})`. Gli interceptor vengono eseguiti nell’ordine in cui sono definiti nell’annotation.

MESSAGE DRIVEN BEAN (MDB)

Un MDB è un componente messo a disposizione da JAVA EE che consente di sviluppare servizi basati sullo scambio di messaggi.

Il concetto che è alla base degli MDB è la coda, quindi non abbiamo un classico servizio basato sul classico modello richiesta/risposta, ma abbiamo un'architettura basata su una serie di richieste inviate al servizio, che vengono messi in coda e successivamente elaborate.

Per creare un MDB è necessario creare una classe JAVA annotata con `@MessageDriven` e che implementi l'interfaccia `MessageListener`.

I servizi di mail ed sms, ad esempio, sono servizi che si basano su code, che vengono in seguito smistate per il server destinatario. Infatti, nell'architettura a messaggi:

- un client invia un messaggio al server;
- il server inserisce il messaggio in una coda;
- sulla coda sono in ascolto gli MDB che ricevono il messaggio ed elaborano la richiesta.

In JAVA EE, lo scambio di messaggi tra client e server è gestito attraverso un componente chiamato Java Message Service (JMS).

Gli elementi principali di JMS sono:

- JMS Broker: gestisce lo scambio di messaggi;
- JMS Producer: il client che crea e invia il messaggio;
- JMS Consumer: l'MDB che riceve il messaggio;
- JMS Queue: il canale di comunicazione su cui è in ascolto l'MDB ed è una coda di tipo FIFO (First In First Out), cioè il primo messaggio che arriva è anche il primo messaggio a essere elaborato dall'MDB;
- JMS Message: messaggio inviato dal client al server che contiene la richiesta.

In JMS sono disponibili 6 tipi di interfacce per creare i messaggi, che si differenziano in base al contenuto del messaggio (chiamato payload):

- Message: classe base usata per notifiche che non hanno contenuto;
- TextMessage: il contenuto del messaggio è una stringa;
- BytesMessage: il contenuto del messaggio è un array di bytes;
- StreamMessage: il contenuto del messaggio è una sequenza di tipi primitivi di JAVA;
- MapMessage: il contenuto del messaggio contiene chiavi-valori, dove le chiavi sono stringhe e i valori possono essere di qualsiasi tipo;
- ObjectMessage: il contenuto del messaggio è un oggetto che implementa Serializable.

Con JMS possiamo inviare messaggi:

- ad un destinatario specifico (point-to-point model): in questo modello architetturale, mittente e destinatario sono a conoscenza l'uno dell'altro;
- in broadcast a clienti interessati ad un certo topic (Publish/Subscribe model): più subscribers sono interessati a ricevere messaggi inviati da un publisher che hanno un certo topic. In questo modello architetturale, mittente e destinatari non si conoscono.

L'utilizzo degli MDB non è necessario quando sviluppiamo applicazioni normali. A senso utilizzarli solo nel momento in cui dobbiamo sviluppare applicazioni di grosse dimensioni, che richiedono la gestione di code applicative per l'elaborazione delle richieste da parte dei client.

WEB SERVICE SOAP CON EJB

SOAP (Simple Object Access Protocol) è un protocollo che definisce le regole per lo scambio di messaggi tra software.

Il protocollo SOAP è un protocollo che è indipendente dalla piattaforma, quindi non è legato a JAVA in particolare, ma qualsiasi linguaggio di programmazione (PHP, .NET e così via) lo implementano. Di conseguenza, se due applicazioni sono scritte in due linguaggi di programmazione diversi, comunque possono comunicare mediante questo protocollo.

Il messaggio SOAP è un messaggio di tipo XML e contiene:

- un header: facoltativo, contiene meta-informationi (ad esempio relative alla sicurezza, alle transazioni e così via);
- un body: obbligatorio, contiene il messaggio inviato dal client al server e viceversa.

Per creare un servizio SOAP è sufficiente creare un EJB con l'annotation @WebService sul nome della classe e l'annotation @WebMethod sul metodo da esporre all'interno del web service.

Vediamo un esempio di creazione di servizio SOAP:

```
import javax.ejb.Stateless;
import javax.jws.WebMethod;
import javax.jws.WebService;

@Stateless
@WebService(serviceName = "SOAP")
public class EJB_SOAP { //http://localhost:8080/CORSO_JAVA_EE/webservices/EJB_SOAP?wsdl

    public EJB_SOAP() {
    }

    @WebMethod(operationName = "saluta")
    public String saluta(String nome, String cognome) {
        return "Ciao " + nome + " " + cognome;
    }
}
```

Mediante l'indirizzo http://localhost:8080/CORSO_JAVA_EE/webservices/EJB_SOAP?wsdl è possibile vedere il seguente documento XML inerente al servizio SOAP creato precedentemente:

```
<?xml version='1.0' encoding='UTF-8'?>
<wsdl:definitions xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:tns="http://EJB_SOAP.service/" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:ns1="http://schemas.xmlsoap.org/soap/http" name="SOAP" targetNamespace="http://EJB_SOAP.service/">
    <wsdl:types>
        <xsschema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:tns="http://EJB_SOAP.service/" elementFormDefault="unqualified" targetNamespace="http://EJB_SOAP.service/" version="1.0">
            <xss:element name="saluta" type="tns:saluta"/>
            <xss:element name="salutaResponse" type="tns:salutaResponse"/>
            <xss:complexType name="saluta">
                <xss:sequence>
                    <xss:element minOccurs="0" name="arg0" type="xs:string"/>
                    <xss:element minOccurs="0" name="arg1" type="xs:string"/>
                </xss:sequence>
            </xss:complexType>
            <xss:complexType name="salutaResponse">
                <xss:sequence>
                    <xss:element minOccurs="0" name="return" type="xs:string"/>
                </xss:sequence>
            </xss:complexType>
        </xsschema>
    </wsdl:types>
    <wsdl:message name="saluta">
        <wsdl:part element="tns:saluta" name="parameters">
    </wsdl:part>
    </wsdl:message>
    <wsdl:message name="salutaResponse">
        <wsdl:part element="tns:salutaResponse" name="parameters">
    </wsdl:part>
    </wsdl:message>
    <wsdl:portType name="EJB_SOAP">
        <wsdl:operation name="saluta">
            <wsdl:input message="tns:saluta" name="saluta">
        </wsdl:input>
            <wsdl:output message="tns:salutaResponse" name="salutaResponse">
        </wsdl:output>
        </wsdl:operation>
    </wsdl:portType>
    <wsdl:binding name="SOAPSoapBinding" type="tns:EJB_SOAP">
        <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
        <wsdl:operation name="saluta">
            <soap:operation soapAction="" style="document"/>
            <wsdl:input name="saluta">
                <soap:body use="literal"/>
            </wsdl:input>
            <wsdl:output name="salutaResponse">
                <soap:body use="literal"/>
            </wsdl:output>
        </wsdl:operation>
    </wsdl:binding>
    <wsdl:service name="SOAP">
        <wsdl:port binding="tns:SOAPSoapBinding" name="EJB_SOAPPort">
            <soap:address location="http://localhost:8080/CORSO_JAVA_EE/webservices/EJB_SOAP"/>
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

Questo documento XML si chiama WSDL (Web Services Description Language), ed è in grado di descrivere le funzioni e i parametri di un web service. Questo documento è quindi una descrizione di come interagire con il servizio in questione ed è usato per la creazione dei client dei servizi web.

Per utilizzare il WSDL, è sufficiente copiarne il contenuto e incollarlo in un file con formato .wsdl.

Per utilizzare un servizio SOAP creato, ci sono due alternative:

- utilizzare Postman come client, importando al suo interno il file WSDL;
- creare un client da 0.

Utilizzando Postman (dopo aver importato il file WSDL) con il servizio SOAP creato precedentemente, otteniamo il seguente risultato:

The screenshot shows the Postman interface for a SOAP request. The request URL is `POST {{EJB_SOAPPorBaseUrl}}/Corso_JAVA_EE/webservices/EJB_SOAP`. The Body tab displays the XML message sent to the service, which includes two parameters: `<arg0>Paolo</arg0>` and `<arg1>Preite</arg1>`. The response tab shows the XML message returned by the service, which includes a `<return>Ciao Paolo Preite</return>` element.

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
3   <soap:Body>
4     <saluta xmlns="http://EJB_SOAP.service/">
5       <arg0>Paolo</arg0>
6       <arg1>Preite</arg1>
7     </saluta>
8   </soap:Body>
9 </soap:Envelope>
10
```

```
1 <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
2   <soap:Body>
3     <ns2:salutaResponse xmlns:ns2="http://EJB_SOAP.service/">
4       <return>Ciao Paolo Preite</return>
5     </ns2:salutaResponse>
6   </soap:Body>
7 </soap:Envelope>
```

WEB SERVICE RESTful CON EJB

REST (REpresentational State Transfer) è un'architettura software (parallela al protocollo SOAP) che consente di erogare servizi per lo scambio di messaggi tra due o più applicazioni, o anche per la semplice visualizzazione di una serie di informazioni.

Come il protocollo SOAP, anche REST è generalmente implementato sul protocollo HTTP.

REST definisce come creare le URL dei servizi e come utilizzare i metodi HTTP (GET per il recupero di informazioni, POST, PUT, PATCH, DELETE per la modifica).

Il principio fondamentale dell'architettura REST è la risorsa, accessibile tramite una URI. Questo è un concetto molto simile a quello che abbiamo visto con le servlet, dove ogni servlet ha un determinato path, per cui analogamente ogni servizio REST risponde ad una determinata URI.

L'URI deve essere strutturata seguendo un determinato criterio, come i seguenti esempi:

Esempi di URL (HTTP GET method...)

- <http://miaapp.it/api/rest/ordine/1234>
- <http://miaapp.it/api/rest/ordini/0/20>
- <http://miaapp.it/api/rest/cliente/info/333>

Per creare un servizio REST è sufficiente creare un EJB con l'annotation `@Path` sul nome della classe, che specifica il path a cui risponde L'EJB.

I metodi che devono essere esposti tramite il servizio REST devono:

- essere annotati con (`@GET`, `@POST`, `@PUT`, ...) a seconda del tipo di operazione che effettua il metodo;
- essere annotati con l'annotation `@Produces` che indica l'output che il metodo produce (ad esempio testo semplice, JSON, XML, ...).
- essere annotati con l'annotation `@Path`, che viene aggiunta all'URI principale per richiamare il metodo specifico all'interno del servizio.

Vediamo un esempio di creazione di servizio REST:

```
import javax.ejb.Stateless;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;

@Stateless
@Path(value="/fattura")
public class EJB_REST { //http://localhost:8080/CORSO_JAVA_EE/fattura

    public EJB_REST() {
    }

    @GET
    @Produces(value="text/plain")
    @Path(value="generaNumero")
    public String generaNumero(){ //http://localhost:8080/CORSO_JAVA_EE/fattura/generaNumero
        return "1234";
    }
}
```

```
@GET  
@Produces(value="text/plain") //http://localhost:8080/CORSO_JAVA_EE/fattura/generaAnno  
@Path(value="generaAnno")  
public String generaAnno(){  
    return "2017";  
}  
}
```

Rispetto a SOAP, non abbiamo la necessità di creare un client da 0, ma può essere invocato anche tramite un normale browser (o anche Postman), perché risponde ad una URI. Vediamo di seguito il risultato dell'esempio tramite browser:

