

Corso Java

16/01/2018

redatto da:
Giuseppe ing. Grosso
16 gennaio 2018

Indice generale

Lezione1.....	3
installazione ambiente di sviluppo.....	3
Installazione eclipse oxygen.....	3
installazione java cm (jdk 8).....	3
Differenze tra jdk e jre.....	5
Esempio di programma in java con metodo main.....	5
Compilazione ed esecuzione in eclipse.....	5
Tipi di dato.....	5
Concetto di Classe o oggetto e istanza.....	6
Programmazione strutturata: interfaccia e classe astratta.....	6
Lezione 3.....	7
Lezione 4.....	7
Lettura di singoli caratteri da file.....	8
Scrittura di caratteri su file.....	9
I/O bufferizzato e formattato.....	9
Lettura di dati da file.....	10
Esercizi.....	10
Lezione 5: introduzione al PDF.....	10
Lezione 6.....	11
Es. di connessione alla base dati.....	12
creazione nuova tabella.....	13
Inserimento in tabella.....	14
Cancellazione valore.....	15
Aggiornamento valori.....	16

Lezione1

installazione ambiente di sviluppo.

Per poter sviluppare in java occorre effettuare alcune installazioni che di seguito vengono evidenziate:

installazione IDE di sviluppo: eclipse (oxygen) release a giugno 2017.

Installazione eclipse oxygen.

Collegarsi al sito:

<http://www.eclipse.org/downloads/eclipse-packages/>

e scaricare il pacchetto



Effettuato il download effettuare doppio click sul pacchetto scaricato ed installarlo.

installazione java cm (jdk 8).

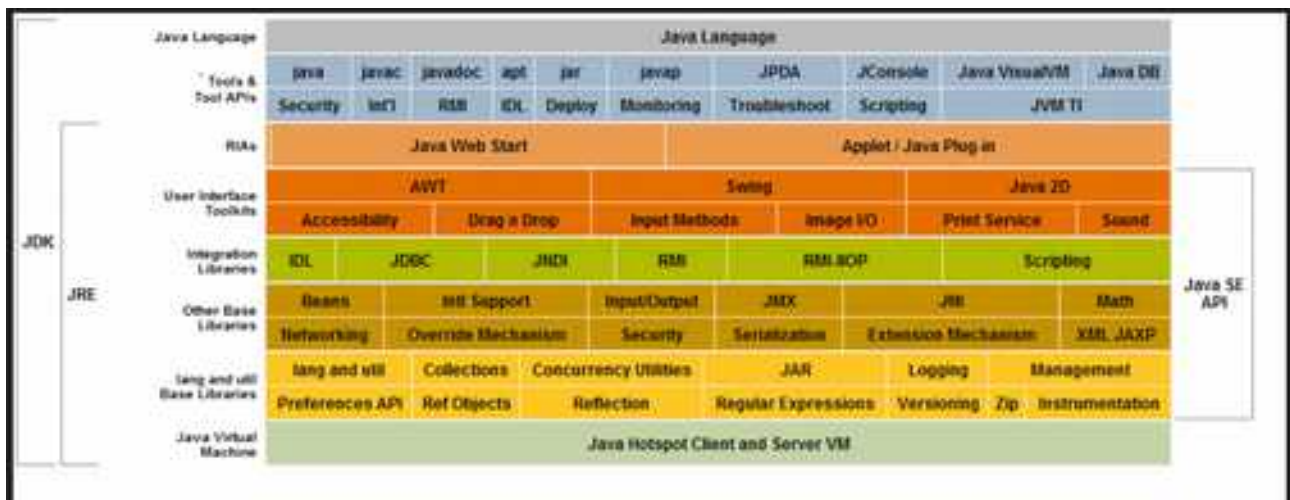
Se non presente la jdk, occorre scaricarla dal sito di oracle:

<http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>



Selezionare la versione di java SE in base al proprio sistema operativo e procedere con l'installazione.

Differenze tra jdk e jre.



La **JDK** è una **estensione della JRE** che contiene questa, ma è inoltre un insieme di software che possiamo utilizzare per sviluppare le applicazioni basate su Java. La Java Development Kit (JDK) è necessaria per sviluppare nuove applicazioni Java.

La differenza principale con la JRE è che contiene **un compilatore Java** per trasformare i nostri codici java in bytecode, cioè in file di tipo .class. E' possibile richiamare il compilatore Java tramite il comando **javac** da linea di comando.

Esempio di programma in java con metodo main.

In questo esempio vediamo come effettuare una semplice classe java con metodo main, ovvero richiamabile ed eseguibile direttamente all'interno dell'ambiente di sviluppo nell'ide eclipse.

Compilazione ed esecuzione in eclipse.

La compilazione del programma in eclipse è molto semplice occorre semplicemente impostare la compilazione automatica e successivamente eseguire il programma java direttamente attraverso il metodo main.

Tipi di dato.

I tipi di dati in java possono essere primitivi (vedi tipi primitivi di seguito) oppure oggetti, sia oggetti nativi che custom (vedi classi).

Fonte wikipedia.

Tipi primitivi [\[modifica \]](#)

I tipi primitivi del Java sono i seguenti:

- *boolean*: ammette i soli valori *true* e *false*.
- Tipi numerici a virgola mobile: *float* e *double*. Essi rispondono alle indicazioni dello standard [IEEE 754](#).
- Tipi numerici interi:

byte	Da -2^7 a $2^7 - 1$	Da -128 a 127
short	Da -2^{15} a $2^{15} - 1$	Da -32768 a 32767
char	Da 0 a $2^{16} - 1$	Da 0 a 65535, ovvero da <code>'\u0000'</code> a <code>'\uffff'</code>
int	Da -2^{31} a $2^{31} - 1$	Da -2147483648 a 2147483647
long	Da -2^{63} a $2^{63} - 1$	Da -9223372036854775808 a 9223372036854775807

Concetto di Classe o oggetto e istanza.

Una classe può essere vista come un contenitore all'interno della quale troviamo due elementi principali: **attributi** e **metodi**.

Gli attributi della classe sono quelle variabili che identificano la classe e che vengono usati dai vari metodi per compiere qualche operazione.

I metodi sono delle funzioni che espletano un certo compito.

L'istanza della classe è una realizzazione fisica della classe, quindi ad esempio due istanze sono completamente indipendenti l'una dall'altra e quindi se creiamo due oggetti di tipo "Numeri" e richiamiamo il metodo di assegnazione passando numeri diversi otterremo due oggetti diversi. Dunque la somma e la sottrazione relative ai due oggetti saranno generalmente diversi.

Programmazione strutturata: interfaccia e classe astratta.

Fino ad adesso abbiamo visto alcuni esempi di implementazioni di classi semplice, proviamo ad analizzare scenari più complessi e a dare una struttura alla programmazione introducendo il concetto di interfaccia e di classe astratta.

Molto spesso le interfacce vengono confuse con le classi astratte dato che dal punto di vista logico sono molto simili, ma le interfacce possono essere intese come un'evoluzione delle classi astratte e permettono, di fatto, di simulare l'ereditarietà multipla.

Il vantaggio principale di una interfaccia, essendo comunque una classe anche se con proprietà particolari, è quello che oltre ad essere estesa può essere **implementata**. La differenza tra estendere e implementare è molto grande in quanto una classe può essere ereditata solo ed esclusivamente una volta, mentre una classe può implementare infinite interfacce permettendo così **l'ereditarietà multipla**.

Le **classi astratte in Java** sono utilizzate per poter dichiarare caratteristiche comuni fra classi di una determinata gerarchia. Pur definendo il nome di un tipo, la classe astratta non può essere istanziata, analogamente a quanto accade per le interfacce; ma a differenza di una interfaccia può avere field non statici, metodi non pubblici, un costruttore, insomma una classe a tutti gli effetti ma non istanziabile.

Lezione 3.

Viene riepilogato quanto già fatto nelle precedenti lezioni:

Definizione di Classe, definizione di interfaccia, definizione di classe astratta.

Esempi su questi tipi di oggetti e tipi di gerarchia tra i vari oggetti.

Metodi privati, protetti, pubblici, statici e finali.

es. di metodi statici il metodo main.

I metodi: parametri di metodo (passaggio per valore)

es. `public void print(String messaggio)`

Concetto di overloading. (stesso nome di metodo con parametri diversi).

es. `print(String messaggio, int code);`

commenti sulle proprietà, sulle classi etc.

Concetti di ereditarietà: classi, superclassi e sottoclassi

es. `public class MyClass extends ParentClass`

e.s `public class ChildrenClass extends MyClass.`

Lezione 4.

Classi per lettura e scrittura di file

Nei programmi sviluppati sino ad oggi abbiamo usato

- output su schermo (con `System.out.print`, `System.out.println` e `System.out.printf`)

- input da tastiera (con la classe Input)

Se un programma deve leggere o scrivere grandi quantità di dati, è conveniente memorizzare questi dati in files.

Java fornisce classi e metodi per scrivere e leggere dati da files (nel package java.io)

I dati possono essere memorizzati in un file in formato

- testo (sequenza di caratteri, leggibile da esseri umani)
- binario (sequenza di byte)

Classi per I/O su files	Input da file	Output su file
Formato testo	FileReader	FileWriter
Formato binario	FileInputStream	FileOutputStream

Queste classi gestiscono I/O di caratteri o bytes da files: per dati più complessi (stringhe, numeri) introdurremo altre classi.

Adesso esaminiamo I/O in formato testo, ma quello in formato binario è del tutto analogo.

I file di testo possono essere aperti, esaminati e modificati usando normali editor (es. emacs). Per visualizzarne il contenuto potete anche usare il comando less della shell di Linux.

Lettura di singoli caratteri da file

Per leggere dati (un carattere alla volta) da un file, occorre, ricordarsi di importare il pacchetto java.io, creare un oggetto della classe FileReader, passando il nome del file al costruttore;

es.

```
FileReader filein = new FileReader("dati.txt");
```

utilizzare (anche più volte) il metodo `public int read()` della classe FileReader per leggere i caratteri;

```
int codiceCarattere = filein.read();
```

infine chiudere il file con il metodo `public void close()`.

```
filein.close();
```

I metodi e il costruttore di FileReader possono lanciare una eccezione di tipo IOException che

rappresenta un errore di I/O. Queste eccezioni sono controllate, e quindi devono essere previste dal programmatore.

Il metodo `read()` restituisce un intero che può essere:

- 1 se si è arrivati alla fine del file;

un intero tra 0 e 16383, che rappresenta il codice di un carattere UNICODE.

Tipicamente, si controlla se il numero letto è diverso da -1 e in questo caso si trasforma l'intero in un `char` usando l'operatore di cast.

Esempio di lettura da file: vedi esempio `FileReadTest.java` lezione4.

Alternativamente si poteva delegare la gestione delle `IOException` al chiamante, aggiungendo la clausola `throws IOException` nell'intestazione del metodo `main`.

Scrittura di caratteri su file

Per scrivere dati su di un file, occorre:

- creare un oggetto della classe `FileWriter`, passando il nome del file al costruttore;
- utilizzare il metodo `public void write(int c)` per scrivere i caratteri;
- chiudere il file con il metodo `close()`.

Anche questi metodi possono lanciare una `IOException`.

Esempio `WriteFileTest.java` che legge da standard input e scrive su file system

Esempio `FileReadWriteTest.java` che legge un file da filesystem e ne scrive uno nuovo.

Se si verifica un'eccezione mentre si tenta di scrivere un carattere nel file, il metodo `close()` non viene eseguito: in certe situazioni questo potrebbe portare alla perdita dei dati scritti nel file.

I/O bufferizzato e formattato

Le classi `FileReader` e `FileWriter` forniscono i metodi basici per leggere o scrivere caratteri su file. Non è conveniente usarle direttamente nei programmi perché non permettono di leggere/scrivere direttamente dati più complessi come stringhe e numeri.

Altre classi di Java forniscono funzionalità di I/O più avanzate, in particolare `BufferedReader` e `BufferedWriter` usano un buffer (memoria tampone) per memorizzare temporaneamente i caratteri

da leggere/scrivere, in modo da ridurre il numero di accessi al file;

PrintWriter fornisce i metodi print e println, che permettono di scrivere qualunque dato Java, convertendolo automaticamente in stringa.

Gli oggetti di queste classi sono dei wrappers: incapsulano gli oggetti delle classi FileReader e FileWriter estendendone le funzionalità.

es. PrintWriterTest.java

Lettura di dati da file

La classe BufferedReader fornisce il metodo readLine() che legge una stringa, ma non ha metodi per leggere, ad esempio, interi o double.

BufferedReaderTest.java: stampa su video il contenuto di un file (FileReadTest.java)

Esempio: somma di interi da file SommaInteriWithException.java

Il programma SommaInteriWithException.java, stampa la somma di una sequenza di interi, contenuti uno per linea nel file integers.txt, con una ragionevole gestione delle eccezioni.

Esercizi

Due semplici esercizi:

Modificare il programma SommaInteriWithException in modo che quando si verifica un'eccezione venga comunque stampata la somma calcolata fino a quel momento.

Modificare il programma SommaInteriWithException in modo che quando si verifica un'eccezione di tipo NumberFormatException dovuta ad una linea del file non convertibile in intero si scarti tale riga ma si prosegua con il calcolo della somma.

Lezione 5: introduzione al PDF.

I pdf in java vengono scritti utilizzando varie librerie, tra le più famose esiste la libreria Itext, questa permette di generare i pdf, di inserire i contenuti ed eventualmente di rileggerli e di aggiungere ulteriori contenuti.

Vedi esempi nel codice.

Documentazione di riferimento

<https://developers.itextpdf.com>

Vediamo quali sono i passi base per poter effettuare la scrittura di un documento pdf:

step 1: preparazione dell'oggetto document

```
Document document = new Document();
```

In questo caso viene istanziato un oggetto Document tramite il suo costruttore vuoto, è possibile istanziare un documento impostando una misura diversa da quella di default (A4), ad esempio vi sono vari formati sia in portrait che in landscape.

// step 2: si stanzia un pdfWriter che provvede a scrivere su filesystem, o su response il contenuto del pdf es. scrittura su FileSystem, dove filename è il path del file nel nostro disco

```
PdfWriter.getInstance(document, new FileOutputStream(filename));
```

// step 3: apertura del documento con inizio delle operazioni

```
document.open();
```

// step 4: aggiunta di informazioni es. testo al documento

```
document.add(new Paragraph("Benvenuto al corso Aeffegroup di Java"));
```

// step 5: chiusura del documento.

```
document.close();
```

Lezione 6.

Connessione alla base dati

Scaricare i driver jdbc per sqlite

<https://bitbucket.org/xerial/sqlite-jdbc/downloads/>

tutorial per sqlite

<http://www.sqlitetutorial.net/download-install-sqlite/>

interfaccia grafica

<https://sqlitestudio.pl/index.rvt>

Java 7

```
try (Connection conn = this.conn();)
```

```
{
```

```
.....
```

```
} catch (Exception e)
```

```
{
```

```
// log errore
```

```
}
```

```
try {
```

```
}
```

```
catch ()
```

```
{
```

```
}
```

```
finally
```

```
{
```

```
// chiusura risorsa.
```

```
}
```

```
/**
```

```
 * select all rows in the warehouses table
```

```
 */
```

```
public void selectAll()
```

```
{
```

```
    String sql = "SELECT * FROM albums";
```

```
    // try with resource ... java 7.
```

```
    try (Connection conn = this.connect();
```

```
        Statement stmt = conn.createStatement();
```

```
        ResultSet rs = stmt.executeQuery(sql))
```

```
    {
```

```
        // loop through the result set
```

```
        while (rs.next())
```

```
        {
```

```
            System.out.println(rs.getInt("albumid") + "\t" + rs.getString("title") +  
            "\t" + rs.getDouble("ArtistId"));
```

```
        }
```

```

    } catch (SQLException e)
    {
        System.out.println(e.getMessage());
    }
    // superfluo...
    disconnect();
}

```

Creazione della base dati:

Es.

```

* create database.
*
* @param fileName
*     the database file name
*/
public void create()
{

    try (Connection conn = this.connect())
    {
        if (conn != null)
        {
            DatabaseMetaData meta = conn.getMetaData();
            System.out.println("The driver name is " + meta.getDriverName());
            System.out.println("A new database has been created.");
        }
    }

} catch (SQLException e)
{
    System.out.println(e.getMessage());
}

```

```
}
```

Es. di connessione alla base dati.

```
/**
 * Connect to the database
 *
 * @return the Connection object
 */
private Connection connect()
{
    // SQLite connection string
    String url = DB_URL + filename;
    Connection conn = null;
    try
    {
        conn = DriverManager.getConnection(url);
    } catch (SQLException e)
    {
        System.out.println(e.getMessage());
    }
    return conn;
}
```

creazione nuova tabella.

```
/**
 * Create a new table in the test database
 *
 */
```

```

public void createNewTable(String mytable)
{
    // SQLite connection string
    this.tablename = mytable;

    // SQL statement for creating a new table
    String sql = "CREATE TABLE IF NOT EXISTS " + mytable + " (\n" + " id integer
PRIMARY KEY,\n"
                + "      name text NOT NULL,\n" + "      age integer\n" + ");";

    try (Connection conn = this.connect(); Statement stmt = conn.createStatement())
    {
        // create a new table
        stmt.execute(sql);
    } catch (SQLException e)
    {
        System.out.println(e.getMessage());
    }
    System.out.println("Created table " + mytable);
}

```

Inserimento in tabella

```

/**
 * insert list of value
 *
 */
public void insert(List<MyTable> lt)
{
    for (MyTable myTable : lt)
    {
        insert(myTable);
    }
}

```

```

    }

    /**
     * insert single value
     *
     */
    public void insert(MyTable myTable)
    {
        // SQLite connection string
        String sql = "INSERT INTO " + tablename + "(id,name,age) VALUES(?,?,?)";
        try (Connection conn = this.connect(); PreparedStatement pstmt =
conn.prepareStatement(sql))
        {
            pstmt.setInt(1, myTable.getId());
            pstmt.setString(2, myTable.getName());
            pstmt.setInt(3, myTable.getAge());
            pstmt.executeUpdate();
        } catch (SQLException e)
        {
            System.out.println(e.getMessage());
        }

        System.out.println("insert values into table " + tablename + " " +
myTable.getName());
    }

```

Cancellazione valore.

```

    /**
     * cancellazione singoli record.
     *
     */
    public void delete(MyTable myTable)
    {

```

```

        delete(myTable.getId());
    }

    /**
     * Delete a warehouse specified by the id
     *
     * @param id
     */
    public void delete(int id)
    {
        String sql = "DELETE FROM " + tablename + " WHERE id = ?";

        try (Connection conn = this.connect(); PreparedStatement pstmt =
conn.prepareStatement(sql))
        {

            // set the corresponding param
            pstmt.setInt(1, id);

            // execute the delete statement
            pstmt.executeUpdate();

        } catch (SQLException e)
        {
            System.out.println(e.getMessage());
        }

        System.out.println("deleted value " + id);
    }

```

Aggiornamento valori.

```

    /**
     * Update data of a warehouse specified by the id
     *

```



```

* @param id
* @param name
*      name of the warehouse
* @param age
*      capacity of the warehouse
*/

public void update(Integer id, String name, Integer age)
{
    String sql = "UPDATE " + tablename + " SET name = ? , " + "age = ? " + "WHERE
id = ?";

    try (Connection conn = this.connect(); PreparedStatement pstmt =
conn.prepareStatement(sql))
    {

        // set the corresponding param
        pstmt.setString(1, name);
        pstmt.setInt(2, age);
        pstmt.setInt(3, id);

        // update
        pstmt.executeUpdate();
    } catch (SQLException e)
    {
        System.out.println(e.getMessage());
    }
}

```

Introduzione ai servizi WEB.

I servizi WEB sono dei servizi che permettono di poter recuperare i dati da un server effettuando delle interrogazioni tramite il protocollo HTTP.

Esistono due tipi di servizi

- REST

- SOAP

Questi si diversificano tra di loro in quanto l'approccio REST propone una visione del Web incentrata sul concetto di risorsa mentre i SOAP Web Service mettono in risalto il concetto di servizio.

Pertanto :

- Un **Web Service RESTful** è custode di un insieme di risorse sulle quali un client può chiedere le operazioni canoniche del protocollo HTTP;
- Un **Web Service basato su SOAP** espone un insieme di metodi richiamabili da remoto da parte di un client;

L'approccio dei SOAP Web service ha mutuato un'architettura applicativa denominata **SOA**, *Service Oriented Architecture*, a cui si è recentemente contrapposta l'architettura **ROA**, *Resource Oriented Architecture*, ispirata ai principi REST.

Il protocollo **SOAP** (*Simple Object Access Protocol*) definisce una struttura dati per lo scambio di messaggi tra applicazioni, riproponendo in un certo senso parte di quello che il protocollo HTTP faceva già. SOAP utilizza HTTP come protocollo di trasporto, ma non è limitato nè vincolato ad esso, dal momento che può benissimo usare altri protocolli di trasporto.

A differenza di HTTP, però, le specifiche di SOAP non affrontano argomenti come la **sicurezza o l'indirizzamento**, per i quali sono stati definiti standard a parte, nello specifico *WS-Security* e *WS-Addressing*.

Quindi SOAP non sfrutta a pieno il protocollo HTTP, utilizzandolo come semplice protocollo di trasporto. REST invece sfrutta HTTP per quello che è, un protocollo di livello applicativo, e ne utilizza a pieno le potenzialità.

Inoltre i Web Service basati su SOAP prevedono lo standard **WSDL**, *Web Service Description Language*, per definire l'interfaccia di un servizio. Questa è un'ulteriore evidenza del tentativo di adattare al Web l'approccio di interoperabilità basato su chiamate remote. Infatti il WSDL non è altro che un **IDL** (*Interface Description Language*) per un componente software.

Da un lato l'esistenza di WSDL favorisce l'uso di tool per creare automaticamente client in un determinato linguaggio di programmazione, ma allo stesso tempo induce a creare una forte dipendenza tra client e server.

REST non prevede esplicitamente nessuna modalità per descrivere come interagire con una risorsa. Le operazioni sono implicite nel protocollo HTTP. Qualcosa di analogo a WSDL è **WADL**, *Web Application Definition Language*, un'[applicazione XML](#) per definire risorse, operazioni ed eccezioni previsti da un Web Service di tipo REST.

Recentemente per i servizi REST è stato introdotto uno standard di fatto, in particolare con l'utilizzo di openAPI, questo strumento permette di effettuare una descrizione dei servizi in maniera tale da poter generare in automatico, attraverso questo descrittore, sia la parte server che la parte client dei servizi.

<https://www.openapis.org/>

Attraverso un tool online è possibile scrivere tramite la sintassi specifica delle regole di openAPI direttamente il descrittore dei servizi.

Es.

<https://editor.swagger.io/>

Es. di servizi REST:

Accesso a github.

<https://api.github.com>

<https://github.com/bluviolin/TrainMonitor/wiki/API-del-sistema-Viaggiatreno>

Come si sviluppano in java i servizi REST.

In java vi sono varie librerie utili per lo sviluppo di servizi REST tra i più diffusi vi sono i seguenti:

- restEasy (<http://resteasy.jboss.org/>)
- jersey (<https://jersey.github.io/>)
- spring (<https://spring.io/guides/gs/rest-service/>)

Questi sfruttano le specifiche JAX-RS che permette tramite annotazioni di invocare le risorse per ottenere i dati relativi. https://en.wikipedia.org/wiki/Java_API_for_RESTful_Web_Services

Le principali specifiche riguardano i protocolli di richieste:

GET, POST, PUT, DELETE

GET → utilizzate per le interrogazioni (che non modificano i dati).

POST → utilizzate per gli inserimenti

PUT → utilizzate per gli aggiornamenti

DELETE → utilizzate per le cancellazioni.

Produces, Consumes che permette di definire il media type per produzione e consumo dei servizi es.

Json/xml tra i più diffusi.

Esempio di applicazione server per i servizi.

Nell'esempio jiraProject viene evidenziato come costruire una applicazione web per erogazione dei servizi tramite il protocollo REST.

Esempio di client per interrogazione dei servizi.

Nella classe AuthenticationTest viene evidenziato un client che si collega a github per il recupero dei dati dai servizi REST.

Lezione 13:

Ripasso dei concetti di Classe.

Ripasso del concetto di metodo Main

Ripasso delle interfacce e classi astratte (ereditarietà).

Costruzione di un progetto java su eclipse.

Gestione degli errori in java: Exception.

Il concetto di exception è un concetto che vi è in qualsiasi linguaggio di programmazione, questo rappresenta un evento che interviene nell'esecuzione di un programma che sfugge dalla normale esecuzione del programma stesso.

Pertanto all'interno del programma occorre contornare le istruzioni che possono produrre una eccezione (o più di una) tramite la sintassi

try..catch..finally.

Oppure tramite la semantica throws nel metodo che si sta implementando, questa ipotesi prevede che il chiamante del metodo debba gestire l'errore.

es.

try

{

... istruzione che può generare una exception

}

catch(Exception e)

{

.. viene catturata l'eccezione, si procede con alcune azioni ed eventualmente si può rilanciare una

eccezione al chiamante oppure anche non fare niente.

```
}
```

```
// opzionale: se presente viene eseguita sempre anche in caso di NON eccezione
```

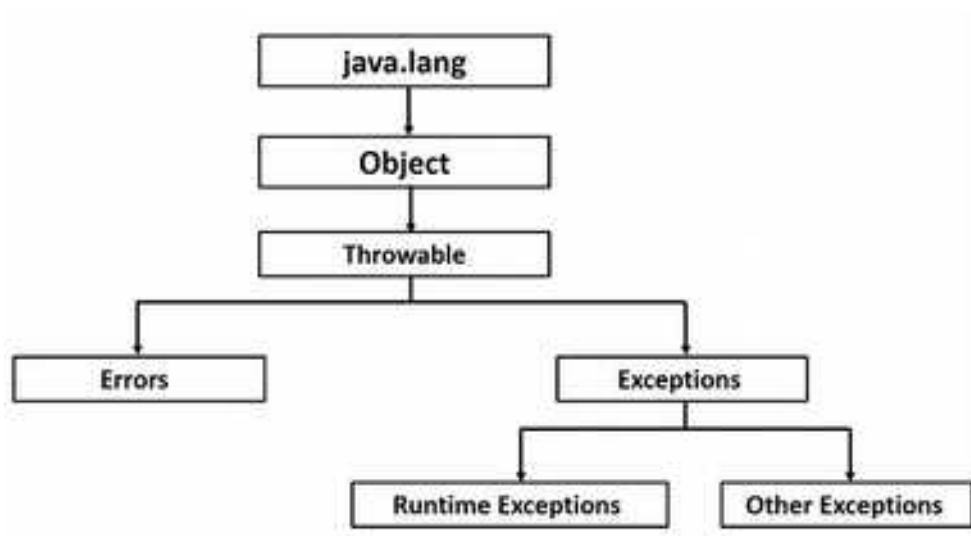
```
finally
```

```
{
```

```
.. effettua alcune azioni.
```

```
}
```

Di seguito la gerarchia delle eccezioni in java:



Exceptions: Metodi:

di seguito i metodi che possiamo invocare per le eccezioni:

Metodo e descrizione

public String getMessage()

1

Ritorna il dettaglio del messaggio dell'eccezione che è intervenuta. Il messaggio è inizializzato dal costruttore della classe Throwable.

public Throwable getCause()

2

Ritorna la causa dell'eccezione presente nell'oggetto Throwable.

3

public String toString()

Ritorna il nome della classe concatenato dal messaggio (getMessage()).

public void printStackTrace()

- 4 Stampa il risultato di toString() con lo stackTrace nel system.err (Error outpustream).

public StackTraceElement [] getStackTrace()

- 5 Ritorna un array che contiene ciascun elemento dello stackTrace. L'elemento all'indice 0 rappresenta il vertice dello stack e l'ultimo quello in basso della catena dello stack.

public Throwable fillInStackTrace()

- 6 Riempie lo stack trace dell'oggetto throwable con lo stack trace corrente, con altre informazioni dello stack trace.

Blocchi fondamentali delle eccezioni: try, catch, finally.

I blocchi fondamentali della eccezione sono tre:

try

{

.. raccoglie le istruzioni che possono generare una eccezione.

}

catch

{

... raccoglie un evento di eccezione e la gestisce.

}

finally

{

... esegue un'azione che viene eseguita sempre sia in caso di eccezione che non.

}

Eccezioni tipiche:

ArrayoutofboundException

NullpointException

FilenotFoundException

etc.

Estensione delle eccezioni.

E' possibile creare delle proprie eccezioni personalizzate che permettono di gestire delle casistiche delle proprie applicazioni.

Es.

AeffeException extends Exception.

In questo modo la nuova eccezione di nome "AeffeException" permette di estendere l'eccezione di tipo Exception, da tale estensione successivamente è possibile intercettare determinati eventi che provocano tale eccezione e gestirla.

es.

```
public class AeffeExceptionMain
{
    AeffeException newExc = new AeffeException("Messaggio della mia eccezione.");

    public AeffeExceptionMain() throws AeffeException
    {
        String msg= "Questa è un esempio di eccezione";
        System.out.println(msg);
        throw newExc;
    }

    public static void main(String args[])
    {
        try
        {
            AeffeExceptionMain use = new AeffeExceptionMain();
        } catch (AeffeException myEx)
        {
            System.out.println("Questa è la mia eccezione di tipo AEFFE:" +
myEx.getMessage());
        }
    }
}
```