UNIVERSITÀ DEGLI STUDI DI SALERNO
DIPARTIMENTO DI INFORMATICA
DIPARTIMENTO DI ECCELLENZA

Università di degli Studi di Salerno
Dipartimento di Informatica

Programmazione ad Oggetti

a.a. 2022-2023

Gestione delle eccezioni

Docente: Prof. Massimo Ficco

E-mail: mficco@unisa.it

Considerazioni

C ed altri linguaggi presentano molti schemi per la gestione degli errori che finiscono con l'essere solo convenzioni e non parte del linguaggio.

Tipicamente si usa ritornare un valore o settare un flag e si suppone che il destinatario controllerà per verificare che l'operazione effettuata sia andata a buon fine.

Tuttavia quando un programmatore utilizza una libreria è propenso a credere che il suo codice sia immune da errori e non effettua tali controlli

Quando gestire gli errori

Il momento ideale in cui catturare un errore è a tempo di compilazione, prima cioè di eseguire il programma.

Non tutti gli errori possono essere individuati a tempo di compilazione …

Il resto dei problemi deve essere gestito a tempo di esecuzione (dipendente dalle condizioni in cui un'applicazione viene eseguita)

Inoltre, errori non gestiti possono manifestarsi come risultati non corretti, o come comportamento non previsto di un programma.

Errori

- Alcune frequenti cause di errori :
  - Memory errors (i.e. memoria non allocata correttamente, memory leaks, “null pointer”);
  - File system errors (i.e. disk full, file non presente);
  - Network errors (i.e. network disconnessa, URL inesistente);
  - Calculation errors (i.e. divisione per zero 0);
  - Array errors (i.e. accesso ad un elemento in posizione –1);
  - Conversion errors (i.e. convertire ‘q’ a un numero).

Esempio: divisione

Un semplice esempio di eccezione è la divisione per zero.

Chi sta effettuando una divisione deve controllare che il denominatore sia diverso da zero.

Chi implementa la divisione può non sapere cosa fare se il denominatore è zero perché non conosce il contesto.

In tal caso tutto ciò che può fare è generare un'eccezione piuttosto che continuare .

Cercare un eccezione

Quando si genera un eccezione accadono diverse cose.

- Prima di tutto viene creato un oggetto Exception come un qualsiasi oggetto Java (new Exception()).
- Quindi viene fermato il percorso di esecuzione (quello che non può continuare) e il riferimento all'oggetto eccezione creato viene espulso dal contesto corrente.
- A questo punto il meccanismo di gestione delle eccezioni comincia a cercare un punto dal quale riprendere l'esecuzione del programma.
- Tale punto è detto *exception handler,* il cui compito è recuperare lo stato corretto del problema così che il programma può percorrere un'altra via o continuare.
- Si dice che il codice che individua l'errore solleva (**throw**) tale oggetto.

Catturare un eccezione

Prevediamo due modalità per catturare l'eccezione:

- Il nostro stesso codice genera un eccezione: usciamo dal metodo.
- Un metodo da noi invocato genera un eccezione

Il codice genera un eccezione

- Un blocco **try** contiene il codice che potenzialmente solleva eccezioni durante la sua esecuzione. Il blocco **try {…}** cattura un'eccezione. Se il l'eccezione viene generata all'interno di un blocco try{…} riusciamo ad evitare l'uscita dal metodo e catturiamo l'eccezione.

- Quello che succede è che non vengono semplicemente completate le restanti istruzioni del blocco a partire dal punto in cui è stata generata l'eccezione.

Exception Handler

- Naturalmente una volta generata l'eccezione occorre definire il punto da cui continuare l'esecuzione.
- Tale punto è detto *exception handler*.
- Esso segue subito il blocco try{..} che cattura l'eccezione.

Esempio

- La gerarchia delle eccezioni è la possibilità di avere più blocchi catch consente di differenziare la gestione delle eccezioni.

```
try {
        // Code that might generate exceptions
} catch(Type1 id1) {
        // Handle exceptions of Type1
} catch(Type2 id2) {
        // Handle exceptions of Type2
} catch(Type3 id3) {
        // Handle exceptions of Type3
}
```

- Un solo handler viene eseguito

L'ordine dipende dalla classe delle eccezioni. Si procede dall'alto verso il basso nell'ordine di specializzazione, dal particolare al generale, mettendo in ultimo il catch(Exception).

Finally

- Una delle differenze più significative tra C++ e Java è che Java supporta la parola chiave finally, per definire un blocco il cui codice viene sempre eseguito indipendentemente dal fatto che il codice nel blocco catch precedente venga eseguito o meno.

```
try { … }
catch (SomeException e) { … }
finally {
  //il codice qui viene sempre eseguito.
}
```

- Lo scopo del blocco finally è di consentire al programmatore la pulizia e il rilascio di risorse, come ad esempio la chiusura di socket, degli handle di file, ecc. Anche se Java esegue un garbage collector, la garbage collection si applica solo alla memoria e a nessun'altra risorsa. Ci sono ancora occasioni in cui il programmatore deve smaltire manualmente le risorse.

Esempio-nofinally

```
public class OnOffSwitch {
private static Switch sw = new Switch();
public static void f() throws OnOffException1,OnOffException2 {}
public static void main(String[] args) {
try {
   sw.on();
   // Code that can throw exceptions...
   f();
   sw.off();
} catch(OnOffException1 e) {
    System.err.println("OnOffException1");
    sw.off();
} catch(OnOffException2 e) {
    System.err.println("OnOffException2");
     sw.off();
}
}
```

Esempio-finally

```
public class WithFinally {
static Switch sw = new Switch();
public static void main(String[] args) {
try {
   sw.on();
   // Code that can throw exceptions...
   OnOffSwitch.f();
} catch(OnOffException1 e) {
   System.err.println("OnOffException1");
} catch(OnOffException2 e) {
   System.err.println("OnOffException2");
} finally {
   sw.off();
}
}
}
```

La clausola catch

- È anche possibile innestare blocchi try-catch all'interno di più blocchi try esterni. In questi casi, un blocco catch interno può inoltrare l'eccezione al suo livello esterno. Questo viene fatto con l'espressione throw; senza argomenti.

- Infatti, una volta intercettata una eccezione, non è raro che il gestore decida di non essere in grado di occuparsi in maniera completa dell'errore che ha scatenato l'eccezione, quindi può decidere di risollevare l'eccezione.

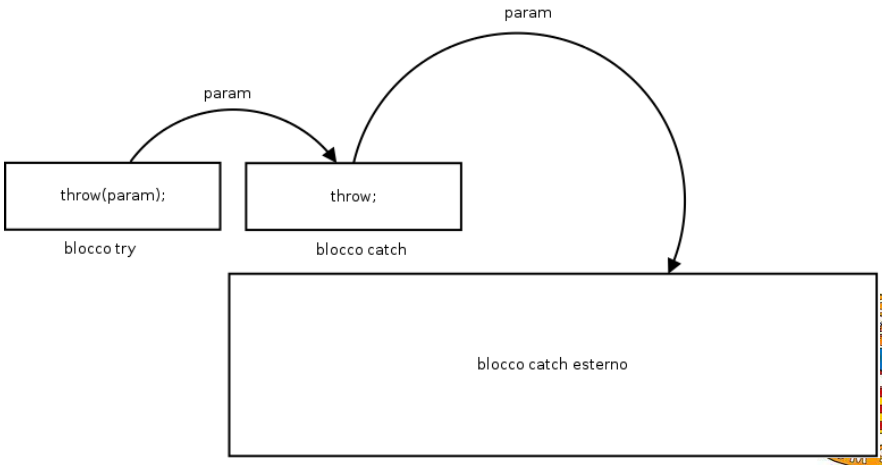
Catch Innestati

```
void h() {
    try { … } catch (Matherr m) {
        if(può_gestirlo_completamente) {
            //gestisce eccezione
            return;
        } else {
            //fa quello possibile localmente
            throw;
    }  catch (…) {
        …
    }
}
```

Catch Innestati

```
void h() {
    try { … } catch (Matherr m) {
        if(può_gestirlo_completamente
            //gestisce eccezione
            return;
        } else {
            //fa quello possibile localmente
            throw;
    }  catch (…) {
        …
    }
}
```

L'eccezione risollevata è quella originale, non solo la parte accessibile come tipo indicato nella clausola catch.



Catch Innestati

```
void h() {
    try { … } catch (Matherr m) {
        if(può_gestirlo_completamente) {
            //gestisce eccezione
            return;
        } else {
            //fa quello possibile localmente
            throw;
    } catch (…) {
        …
    }
}
```

Consente di intercettare tutte le possibili eccezioni.

Metodo che genera un eccezione
Eccezioni - Throw

- Una funzione che può sollevare un'eccezione può specificare ciò nella sua firma:

  void f(int a) **throw (x2, x3);**

Metodo che genera un eccezione
Eccezioni - Throw

- Una funzione che può sollevare un'eccezione può specificare ciò nella sua firma:

```
void f(int a) throw ( x2, x3);
```

Tipi delle eccezioni che possono essere sollevate durante l'esecuzione di f().

*Metodo che genera un eccezione
Eccezioni - Throw

- Una funzione che può sollevare un'eccezione può specificare ciò nella sua firma:

  void f(int a) throw (x2, x3);

- Se durante l'esecuzione della funzione f(), si cerca di violare la specifica della sua firma, il tentativo viene trasformato in una chiamata a std::unexpected(), il cui significato di default è l'invocazione di **std::terminate()** che termina il programma.

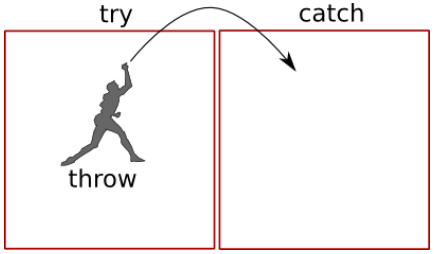
- Se nella firma una funzione non specifica quali eccezioni può sollevare, implicitamente significa che può sollevare qualunque eccezione. Inoltre, tutte le dichiarazioni di una funzione devono possedere esattamente la stessa firma con la specifica delle eccezioni sollevate come la sua definizione, o anche una più restrittiva.

Metodo che genera un eccezione
Eccezioni - Throw

La clausola throw delega

```
/**
 * Calcola la differenza in giorni fra due date, specificate rispettivamente dal giorno
 * "gg1", mese "mm1" e anno "aa1" e giorno "gg2", mese "mm2" e anno "aa2"
*/
public int differenzaDate(int gg1, int mm1, int aa1, int gg2, int mm2, int aa2)
 throws DataNonValida
 {
    if(!dataValida(gg1, mm1, aa1) || !dataValida(gg2, mm2, aa2))
        throw new DataNonValida();
    else {
        int risultato;
        // ... calcola la differenza fra le date
        return risultato;
    }
}
```

- L'effetto di una throw è di srotolare lo stack finchè non viene incontrata una catch appropriata (in una funzione che, direttamente o indirettamente, ha invocato la funzione che ha sollevato l'eccezione).



- Se non viene trovato nessun catch che gestisce l'eccezione il programma si chiuderà, distruggendo (a ritroso) tutti gli oggetti creati dal programma tranne quelli allocati dinamicamente.

Metodo che genera un eccezione
Eccezioni - Throw

Esempio:

```
public void Stampa (String a) throws ErroreStringaVuota /* Significa che  questo
                                                            metodo può lanciare un'eccezione
                                                            di tipo ErroreStringaVuota */

  { if (a == null) throw new ErroreStringaVuota();
        else System.out.println(a); }
```

Quindi a questo punto il metodo Stampa manderà sullo schermo la stringa passatagli come parametro, e genererà una eccezione di tipo ErroreStringaVuota se il puntatore era null. L'eccezione lanciata da noi si dice **eccezione controllata**, questa deve essere obbligatoriamente gestita, mentre quelle lanciate da Java non per forza devono essere catturate e gestite.

- **throws** dichiara che un metodo rilancia all'esterno un eccezione
- **throw** genera (solleva) un eccezione

Creare una propria eccezione

- Posso utilizzare eccezioni predefinite (ex. NumberFormatException), oppure un programmatore può definisce una proprio eccezione a partire dalla classe Exception presente in java.lang.

- Dobbiamo definire due costruttori: quello di default e quello con un parametro stringa

```
class MyException extends Exception {
        public MyException() {}
        public MyException(String msg) { super(msg); }
}
```

Creare una propria eccezione

```
public class SimpleException extends Exception {
        public SimpleException{}
        public SimpleException(String s) {super(s); }}

public class SimpleExceptionDemo {
        public void f(int i) throws SimpleException {
            if (i>0) {
                System.out.println("Throw SimpleException from f()");
                throw new SimpleException(); }} // istanzia un oggetto SimpleException
                // throw new SimpleException("Throw SimpleException from f()"); } // in alternativa

        public static void main(String[] args) {
                    SimpleExceptionDemo sed = new SimpleExceptionDemo();
        try {
                    sed.f(5);
        } catch(SimpleException e) {System.err.println("Caught it!");}
}
```

Eccezioni predefinite

In alcuni casi la JVM genera automaticamente una eccezione:

- Quando si usa un riferimento null *(**NullPointerException**)*
- Quando si accede ad un elemento di un array oltre la sua lunghezza *(**ArrayOutOfBoundException**)*

Si pensi a quanto sarebbe oneroso controllare ogni volta che si usa un riferimento se questo è null

Eccezioni predefinite

Le eccezioni predefinite possono essere catturate e gestite direttamente in in try-catch:

```
…
t=null;
try {
   System.out.println(t.getMessage());   }
catch(NullPointerException  nullPointerException)
            { System.out.println("t=null"); }
```

Eccezioni predefinite

Esistono due costruttori per tutte le eccezioni standard

- Il costruttore di default;
- Il secondo con un argomento che descrive l'eccezione (tipo String).

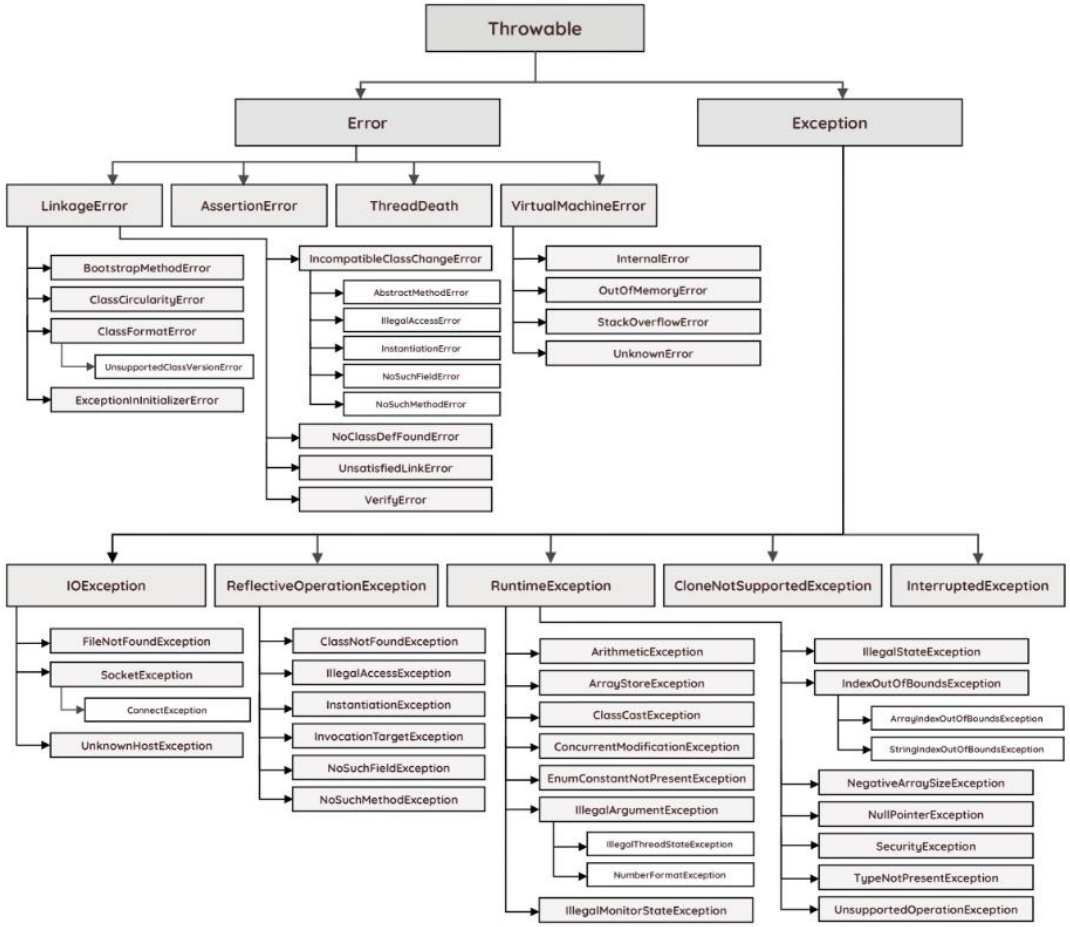
```
if(t == null)
        throw new NullPointerException();
    oppure
        throw new NullPointerException("t = null");
```

Questa informazione può essere recuperata da chi gestirà l'eccezione, per esempio attraverso il catch

```
 ….
} catch (NullPointerException e) {
            System.out.println ("NullPointer Errore:" + e);  }
```

Gerarchia di Eccezioni

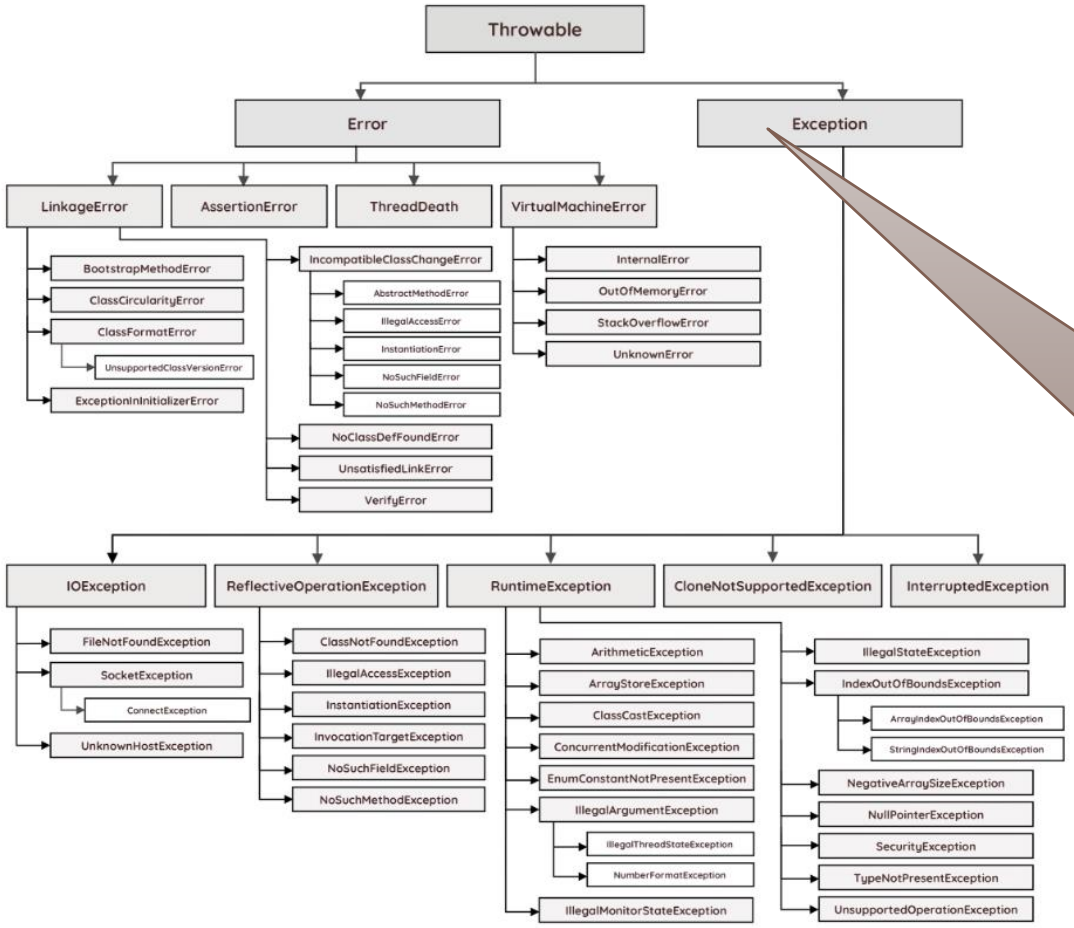
Le eccezioni Java possono essere di diversi tipi e tutti i tipi di eccezione sono organizzati in una gerarchia gen-spec.



- La classe in cima alla gerarchia delle classi di eccezione è la classe Throwable, che è una sottoclasse diretta della classe Object. Throwable ha due sottoclassi dirette: Exception ed Error.

Gerarchia di Eccezioni

Le eccezioni Java possono essere di diversi tipi e tutti i tipi di eccezione sono organizzati in una gerarchia gen-spec.

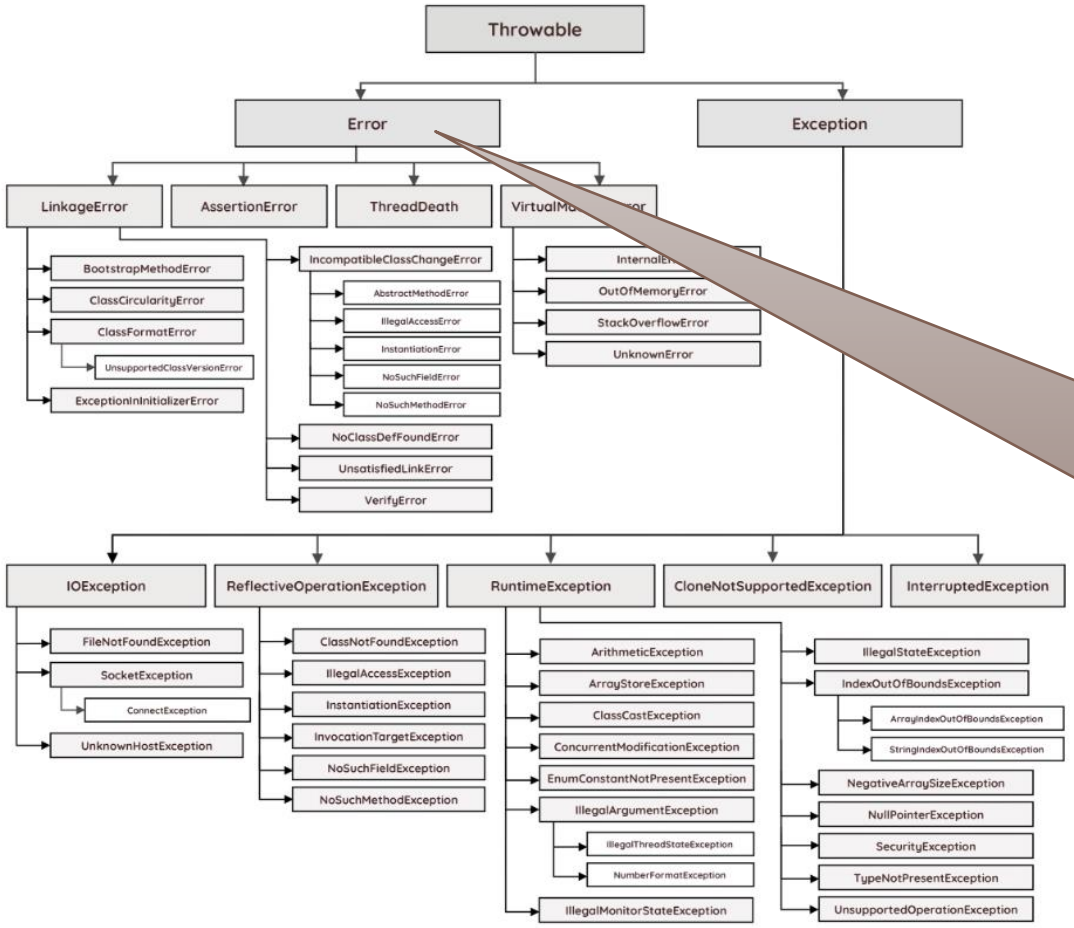


- La classe in cima alla gerarchia delle classi di eccezione è la classe Throwable, che è una sottoclasse diretta della

La classe Exception viene utilizzata per le condizioni di eccezione che l'applicazione potrebbe dover gestire.

Gerarchia di Eccezioni

Le eccezioni Java possono essere di diversi tipi e tutti i tipi di eccezione sono organizzati in una gerarchia gen-spec.



- La classe in cima alla gerarchia delle classi di eccezione è la classe Throwable, che è una sottoclasse diretta della cl

La classe Error viene utilizzata per indicare un problema più serio nell'architettura e non deve essere gestita nel codice dell'applicazione (perchè non è recuperabile).

Eccezioni controllate e non controllate

Le eccezioni si dividono in due categorie:

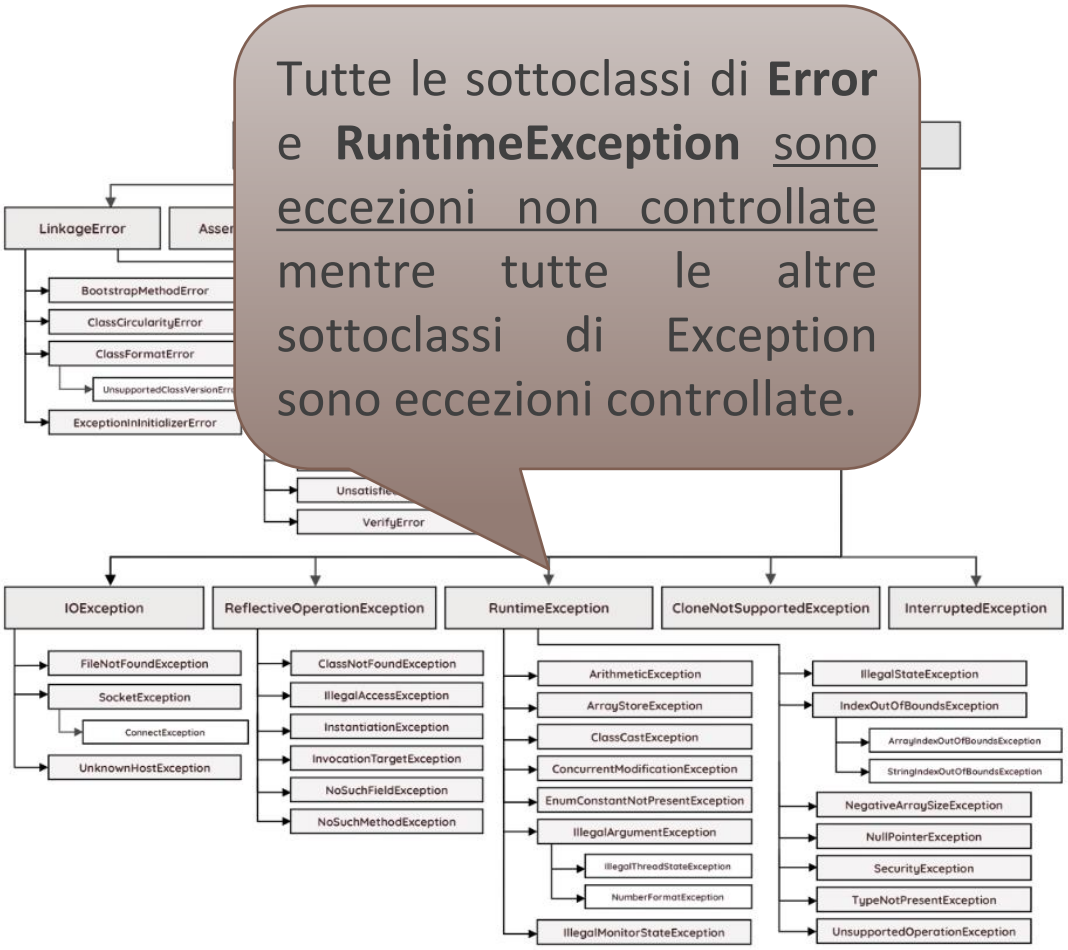
- Eccezioni **controllate** (**checked**)
- Eccezioni **non controllate** (**unchecked**)

Le eccezioni ***controllate*** DEVONO essere gestite esplicitamente dal programma, altrimenti il compilatore segnalerà un errore.

Ogni volta che scriviamo un'istruzione che potrebbe lanciare un'eccezione controllata, allora:

- l'istruzione deve essere racchiusa in un blocco **try-catch** che possa gestire quel tipo di eccezione;
- oppure il metodo che contiene l'istruzione deve ***delegare*** la gestione dell'eccezione al chiamante, con la clausola **throws**.

Gerarchia di Eccezioni



- Sussistono due tipi di eccezioni:
  1. **checked**: condizioni recuperabili e che quindi possono essere gestite dal metodo chiamante;
  2. **unchecked**: condizioni non recuperabili, generalmente dovute ad errori di programmazione e quindi non prevedibili.
     1. Una eccezione di tipo **RunTimeException** può verificarsi ovunque nel programma: un controllo esplicito appesantisce i programmi senza aumentare l'informazione e la robustezza del programma.

Eccezioni Controllate e Non Controllate

- Ecco un esempio di un metodo che genera un errore, che non viene gestito nel codice:

```
public static void print(String myString) {
    print(myString);
}
```

- In questo esempio, il metodo ricorsivo "print" chiama se stesso più e più volte fino a raggiungere la dimensione massima dello stack di thread Java, a quel punto esce con un StackOverflowError:

```
Exception in thread "main" java.lang.StackOverflowError
at StackOverflowErrorExample.print(StackOverflowErrorExample.java:6)
```

- Il metodo genera l'errore durante l'esecuzione ma non lo gestisce nel codice: il programma esce semplicemente quando si verifica l'errore poiché è irrecuperabile e richiede una modifica nel codice stesso.

Eccezioni Controllate e Non Controllate

- Ecco un esempio di un metodo che gestisce un'eccezione controllata:

```
public void writeToFile() {
    try (BufferedWriter bw = new BufferedWriter(new FileWriter("myFile.txt"))) {
        bw.write("Test");
    } catch (IOException ioe) {
        ioe.printStackTrace();
    }
}
```

- In questo esempio, entrambe le istruzioni all'interno del blocco try (l'istanza dell'oggetto BufferedWriter e la scrittura su file utilizzando l'oggetto) e la chiusura della risorsa possono generare **IOException**, che è un'eccezione verificata e pertanto deve essere gestita dal metodo o dal suo chiamante. Nell'esempio, IOException viene gestita all'interno del metodo e l'analisi dello stack dell'eccezione viene stampata sulla console.

Eccezioni Controllate e Non Controllate

- Ecco un esempio di un metodo che genera un'eccezione non controllata (**NullPointerException**) che non viene gestita nel codice:

```
public void writeToFile() {
    try (BufferedWriter bw = null) {
        bw.write("Test");
    } catch (IOException ioe) {
        ioe.printStackTrace();
    }
}
```

- Quando viene chiamato il metodo precedente, viene generata un'eccezione NullPointerException perché l'oggetto BufferedWriter è null. Poiché NullPointerException è un'eccezione non controllata, non è necessario gestirla nel codice: è stata gestita solo l'eccezione verificata (IOException).

Eccezioni Controllate e Non Controllate

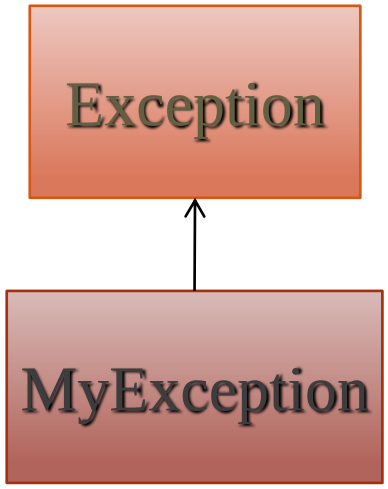
- Ecco un esempio di un metodo che genera un'eccezione non controllata (NullPointerException) che non viene gestita nel codice:

```
public void writeToFile() {
    try (BufferedWriter bw = null) {
        bw.write("Test");
    } catch (IOException ioe) {
        ioe.printStackTrace();
    }
}
```

La clausola throws viene inserita nella dichiarazione del metodo per informare il compilatore che durante l'esecuzione di quel metodo possono essere generate eccezioni (controllate) dei tipi elencati dopo la parola chiave throws, la cui gestione viene delegata al chiamante. In questo esempio non è presente perchè viene sollevata un'eccezione non controllata.

- Quando viene chiamato il metodo precedente, viene generata un'eccezione NullPointerException perché l'oggetto BufferedWriter è null. Poiché NullPointerException è un'eccezione non controllata, non è necessario gestirla nel codice: è stata gestita solo l'eccezione verificata (IOException).

Eccezioni



- Consideriamo il diagramma delle classi a lato in cui un programmatore definisce una proprio eccezione a partire dalla classe Exception presente in java.lang. Inoltre definisce il metodo:

  void do_smtg() throws MyException;

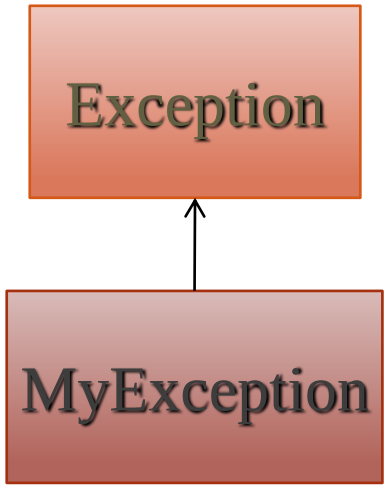
- Consideriamo il seguente frammento di codice:

```
try{
…
do_smtg();
…
} catch(Exception ex) {…}
catch(MyException my_ex) {…}
```

```
import java.lang.Exception;

class MyException extends Exception {
  public MyException() {}
  public MyException(String msg) {
  super(msg); }
}
```

Eccezioni



- Consideriamo il diagram
  programmatore definisc
  classe Exception present
  metodo:

  void do_smtg() thr

- Consideriamo il seguente frammento di codice:

```
try{
…
do_smtg();
…
} catch(Exception ex) {…}
catch(MyException my_ex) {…}
```

Java tenta sempre di eseguire il primo blocco catch in grado di gestire l'eccezione appena sollevata. Il catch di Exception è utilizzabile per risolvere eccezioni di tipo MyException, quindi viene sempre eseguito, mentre l'altro mai.

Metodi della classe Exception

Un eccezione è un oggetto istanza di **java.lang.Throwable** pertanto posso invocare i sui metodi:

- **String getMessage( )** // restituisce la descrizione dell'eccezione
- **String getLocalizedMessage( )**

```
catch (Exception ex) {
            System.out.println(ex.getMessage());
             System.exit(1);
        }
```

Lo StackTrace

- **void printStackTrace( )**
- **void printStackTrace(PrintStream)**
- **void printStackTrace(java.io.PrintWriter)**

```
try {
    f();
} catch(MyException e) {
    e.printStackTrace();}
```

- Il metodo *printStackTrace()* stampa lo stack delle chiamate a procedura a partire dalla classe più esterna fino a quella in cui è stata generata l'eccezione.
- *Fornisce per ogni classe il numero di riga.*
- *Molto più semplice il debug.*

System.err

- **System.err** è lo standard di uscita per gli errori

- E' simile a System.out, ma è usato solo per stampare il testo degli errori
  - System.out.println() stampa sullo standard output (a video e rediretto su file).
    System.err.println() stampa sullo standard error (a video e rediretto su file).

- L'utilità di avere due stream diversi sta nella possibilità di poter redirigire l'output conservando invariato l'errore

System.err

```
try {
    InputStream input = new FileInputStream("c:\\data\\...");
    System.out.println("File opened...");
} catch (IOException e){
        System.err.println("File opening failed:");
        e.printStackTrace(); }
```

Per redirigire su un file:

```
FileOutputStream f = new FileOutputStream("file.txt");
System.setErr(new PrintStream(f));
```

Note: System.setOut() per lo standard output

```
try { InputStream input = new FileInputStream("c:\\data\\..."); System.out.println("File c
```