# Programmazione II

# Gestione delle eccezioni

# Attenzione…

- Questi lucidi sono una semplice TRACCIA per affrontare l'argomento delle ECCEZIONI.
- Esse vanno integrate da un attento studio degli esempi di codice che le accompagnano e NON SOSTITUISCONO UNA ATTENTA LETTURA DEL LIBRO E UN PO' DI ESPERIMENTI DA FARE IN PROPRIO
- Si raccomanda inoltre la lettura del tutorial ufficiale della SUN a riguardo

# Comportamenti Anomali di un Programma

Una classe che nella maggior parte dei casi funziona bene può incappare in una serie di **problemi difficilmente prevedibili**:

- un errore dell'utente (umano o altra classe) nel "comporre" il messaggio che la classe si attende. ESEMPIO: la classe chiede un input numerico e l'utente passa un parametro non valido (es. una stringa vuota).
- un problema generatosi in qualche punto del sistema con cui il codice interagisce (rete/hardware/Sistema Operativo) che difficilmente può essere stimato prima.

- …Ancora
  - Esaurimento memoria
  - Divisioni per zero
  - …
  - Array index "out of bounds"

# Comportamenti Anomali di un Programma

Il punto di vista della programmazione “tradizionale” è che un codice CORRETTO deve in qualche modo occuparsi di prevenire tutte le situazioni anomale e attrezzarsi a gestirle propriamente.

Sebbene tale richiesta abbia senso e sia raccomandabile è spesso IMPOSSIBILE da realizzare, oppure appesantisce la struttura logica del programma che risulta totalmente offuscato da tutti i controlli che si debbono operare per renderlo ROBUSTO.

# Al verificarsi di un errore…

- Si dovrebbe :
  - Ritornare ad una situazione normale e permettere all'utente di eseguire altre operazioni.
  - Notificare l'errore all'utente, salvare le modifiche dei dati e permettere l'uscita dal programma.

- In ogni caso, è necessario un *gestore di errori* che affronti la situazione anomala.

# In passato …

…il modo più comune di operare era quello di restituire un particolare valore convenzionale, nel caso in cui qualcosa fosse andato storto, demandando al processo chiamante il compito di gestire la situazione anomala verificatasi.

I principali limiti di tale approccio sono la ***duplicazione*** di codice per il *check* dell'errore e la conseguente ***illeggibilitá*** complessiva.

# In passato…

Alcuni stili di programmazione prevedono l'uso di "codici di ritorno" (codici di errore) per informare il chiamante sull'esito dell'esecuzione della procedura.

```
int Scrivi (…)
// -1 => errore di scrittura
//  0 => tutto ok
{ … }
```

# In passato…

Tuttavia il chiamante può anche ignorare questi codici, col rischio di far fallire l'esecuzione di tutto il programma.

```
if ( Scrivi(…)== 0 )
{ Bene(); }
else
{ Male(); }
```

```
Scrivi(…);
```

# Cosa è una “eccezione” in un programma?

Java prevede il meccanismo delle eccezioni per gestire le situazioni critiche.

**Una *eccezione* è un evento (eccezionale) che durante l’esecuzione di un programma interrompe il normale flusso di istruzioni.**

Supponiamo che durante l’esecuzione di un certo metodo si verifichi un errore. In tale condizione il metodo crea un “***oggetto eccezione***” che contiene informazioni sul tipo di errore che si è verificato e sullo stato del programma quando tale errore si verifica.

# Cosa è una “eccezione” in un programma?

Tale “***oggetto eccezione***” viene “passato” al *run time system* (ossia l’istanza della JVM).

In termini tecnici tale meccanismo si chiama ***“throwing an exception”***.

Il *run time system* a sua volta cercherà di trovare qualcosa o qualcuno che possa porre rimedio al problema. Se tale “rimedio” non si trova il programma si arresta, altrimenti viene adottato il rimedio.

Ma dove viene cercato chi può (o deve) provvedere all’emergenza?

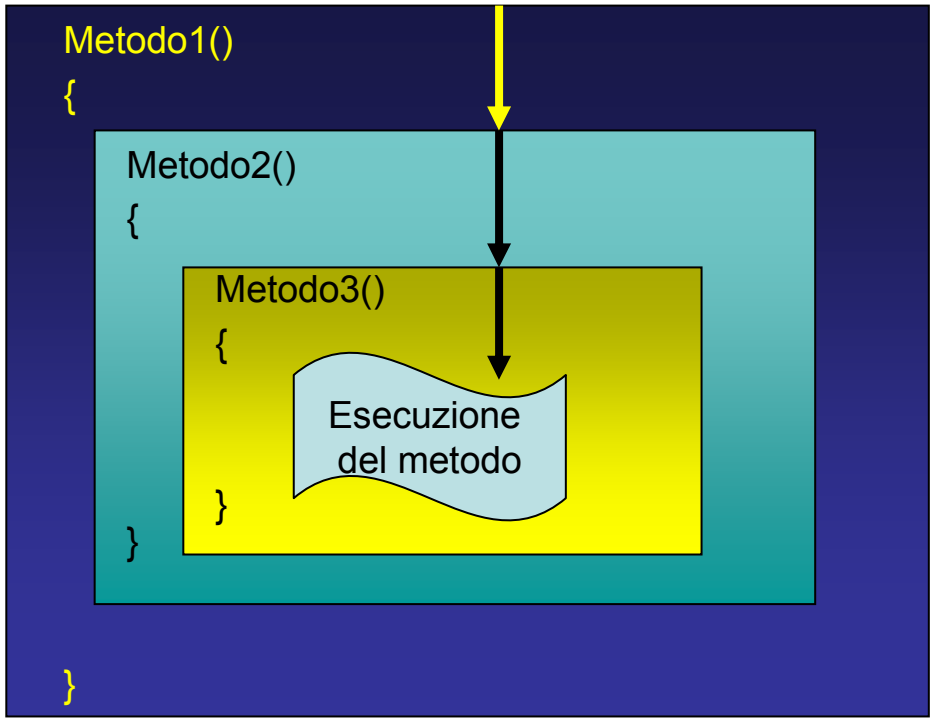
# Gestione delle eccezioni

- Le eccezioni devono essere gestite per fare il recovery da un errore

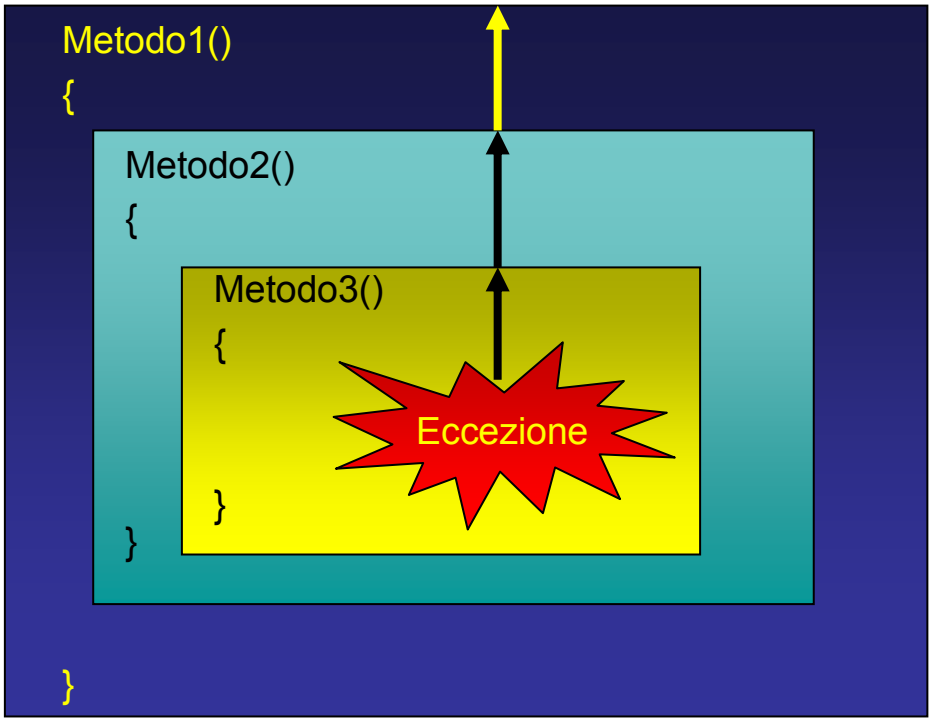
- Come:
  - Processare le eccezioni nei moduli del programma
    - La gestione degli errori che possono presentarsi nel flusso del codice va effettuata nel posto dove questi possono presentarsi
  - Gestire le eccezioni in un modo uniforme

- La gestione delle eccezioni consente di rimuovere il codice della gestione degli errori dalla *"linea principale"* dell'esecuzione del programma

# Propagazione (*throwing*) di un'eccezione…




Sequenza di invocazione
delle chiamate



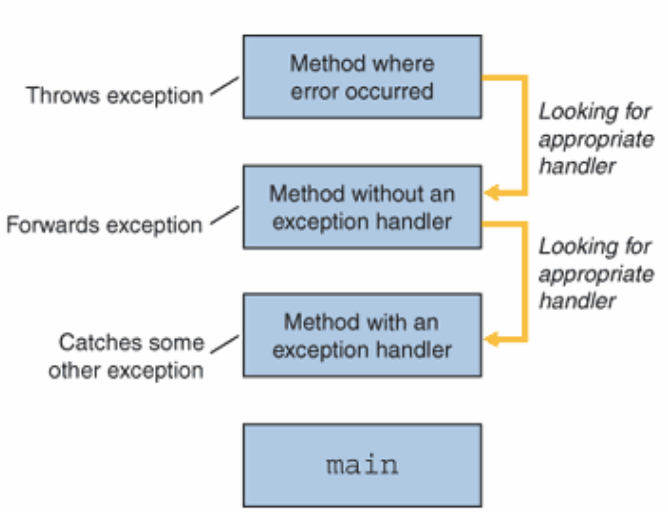

Eccezione nell'ultimo metodo
della sequenza

La propagazione a ritroso va gestita opportunamente.

# La “catena di invocazione dei metodi”

Il codice che deve essere eseguito quando si è verificata l’eccezione, in termine tecnico si chiama “**exception handler**” (gestore dell’eccezione ).

- Quando viene lanciata l’eccezione il metodo controlla se esso stesso possiede un *exception handler*.
- Se non è così cessa la sua attività e restituisce il controllo del flusso del programma al metodo chiamante, se quest’ultimo ha un *exception handler* esso viene attivato, altrimenti ***si procede su su lungo la catena di chiamate fino al main***.
- Se neanche esso ha un *exception handler* il programma termina.




Quando un metodo possiede un *exception handler* per gestire il problema si dice che esso “cattura l’eccezione” (**catch**).

# Exception handler

L'***exception handler*** a seconda dei casi può effettivamente eseguire un *recover* dall'errore (implementando la procedura di recovery) oppure consentire un'uscita pulita dal programma.

Invece di controllare il valore di ritorno di ciascun metodo si può demandare all'*handler* tale operazione che così avviene in maniera del tutto trasparente al programmatore.

Il codice ne guadagna in:

- **Leggibilità**
- **Manutenibilità**

# Checked vs Unchecked exceptions

*Non tutti i problemi che portano all'emissione di eccezioni hanno eguale trattamento da parte del compilatore.*

Alcune situazioni "pericolose" sono in gran parte "prevedibili" o "probabili" in determinate classiche situazioni che possono essere individuate sin dalla compilazione. In questo caso si parla di eccezioni **"checked".**

- **ESEMPI:** non trovare un file nella directory di lavoro, non riuscire a connettersi ad una URL, non ottenere un input dallo "stream" di input, eccetera.

- **Java esige che in tale situazione si implementi un meccanismo di sicurezza, mancando tale meccanismo la compilazione del byte code non va a buon fine.**

# Checked vs Unchecked exceptions

*Non tutti i problemi che portano all'emissione di eccezioni hanno eguale trattamento da parte del compilatore.*

Altre situazioni "pericolose" sono largamente imprevedibili e il compilatore non tenta neanche di individuarle. Esse si manifestano solo a "run time" durante l'esecuzione del programma. In questo caso si parla di eccezioni **"unchecked"**.

- **ESEMPI:** dividere per zero in un'espressione aritmetica, scorrere un array oltre al suo massimo indice, cercare un carattere in una stringa oltre al suo termine, eccetera.

- **Java non richiede che il programmatore scriva un *exception handler* per tali situazioni. Una tale richiesta sarebbe eccessiva e spesso impossibile da soddisfare**

# try…catch

## *(prova a fare questo… e se non funziona rimedia così)*

- Racchiudere il codice che può avere un errore in un blocco **`try`**
- Farlo seguire da uno o più blocchi **`catch`**
  - Ogni blocco catch contiene un exception handler. Possono esserci più catch, purché prevedano eccezioni di tipo differente.

```
try{
   //blocco di istruzioni

}
catch (ExceptionType ref) {
   //codice da eseguire se
   //qualcosa nel blocco try
   //genera un'eccezione del tipo
   //specificato nella clausola "catch"
}
```

  - Se l'eccezione si presenta e per essa esiste il corrispondente gestore specificato nel blocco catch il codice contenuto in esso viene eseguito
  - Se non viene generata nessuna eccezione
    - Il codice di gestione è saltato
    - Il controllo ritorna dopo il blocco catch (l'ultimo)

# Tempo di esempi

- Studiare il codice dell'esempio **L01_01**.
  *Tale codice mostra un codice che prevede una eccezione "unchecked" in un metodo. Essa produce a run time l'arresto irregolare del programma. Notare il messaggio che appare sulla console che riporta l'intera catena di chiamate all'interno delle quali viene tentata la ricerca di un "exception handler".*

- Studiare il codice dell'esempio **L01_02**.
  *Tale codice NON COMPILA. Esso prevede una operazione che potrebbe generare una eccezione prevedibile e richiede che si scriva del codice di "sicurezza". Notare l'errore generato dal compilatore.*

# Tempo di esempi

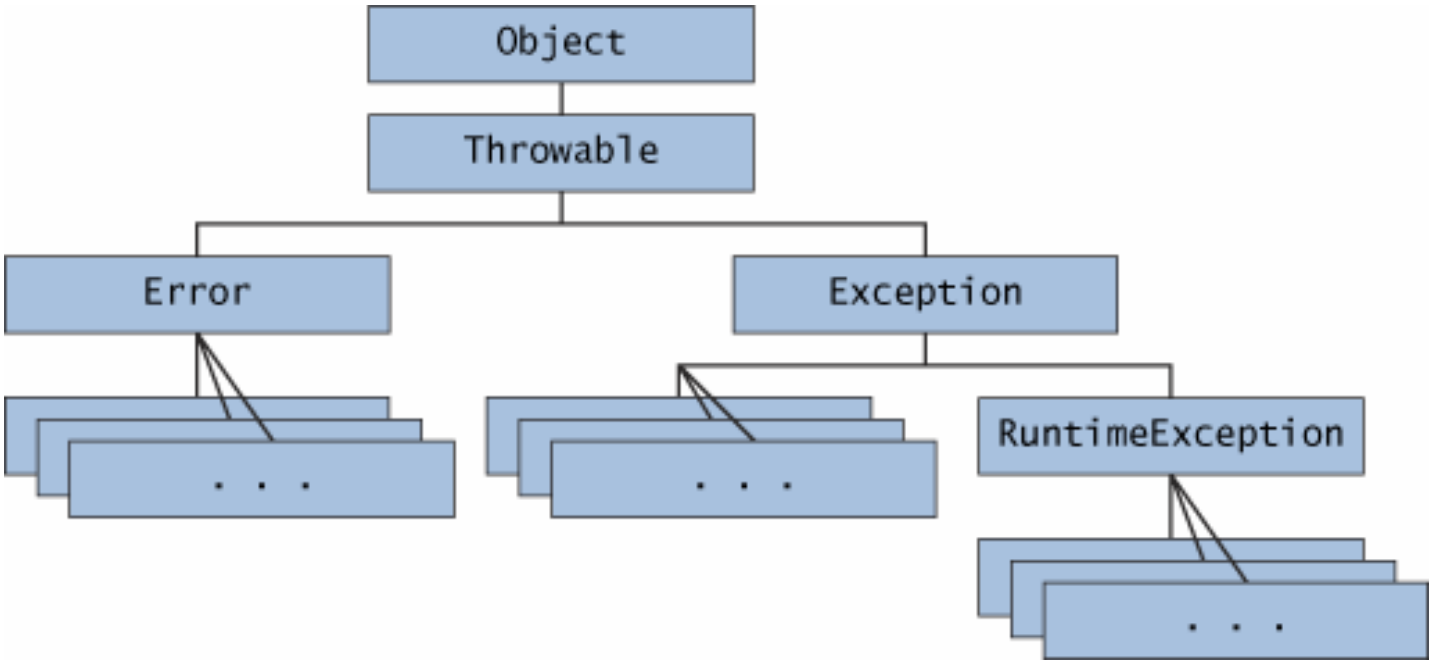
- Studiare il codice dell'esempio **L01_03**.
  *Tale codice è lo stesso dell'esempio L01_02 ma con un exception handler (costrutto "try … catch").*

- Studiare il codice dell'esempio **L01_04**.
  *Tale codice mostra il medesimo codice del L01_01 che prevede una eccezione "unchecked" in un metodo. Sebbene l'eccezione non obblighi il programmatore a creare un "exception handler" in questo caso una clausola "try … catch" è stata prevista. Non è obbligatoria ma non è vietata!!!*

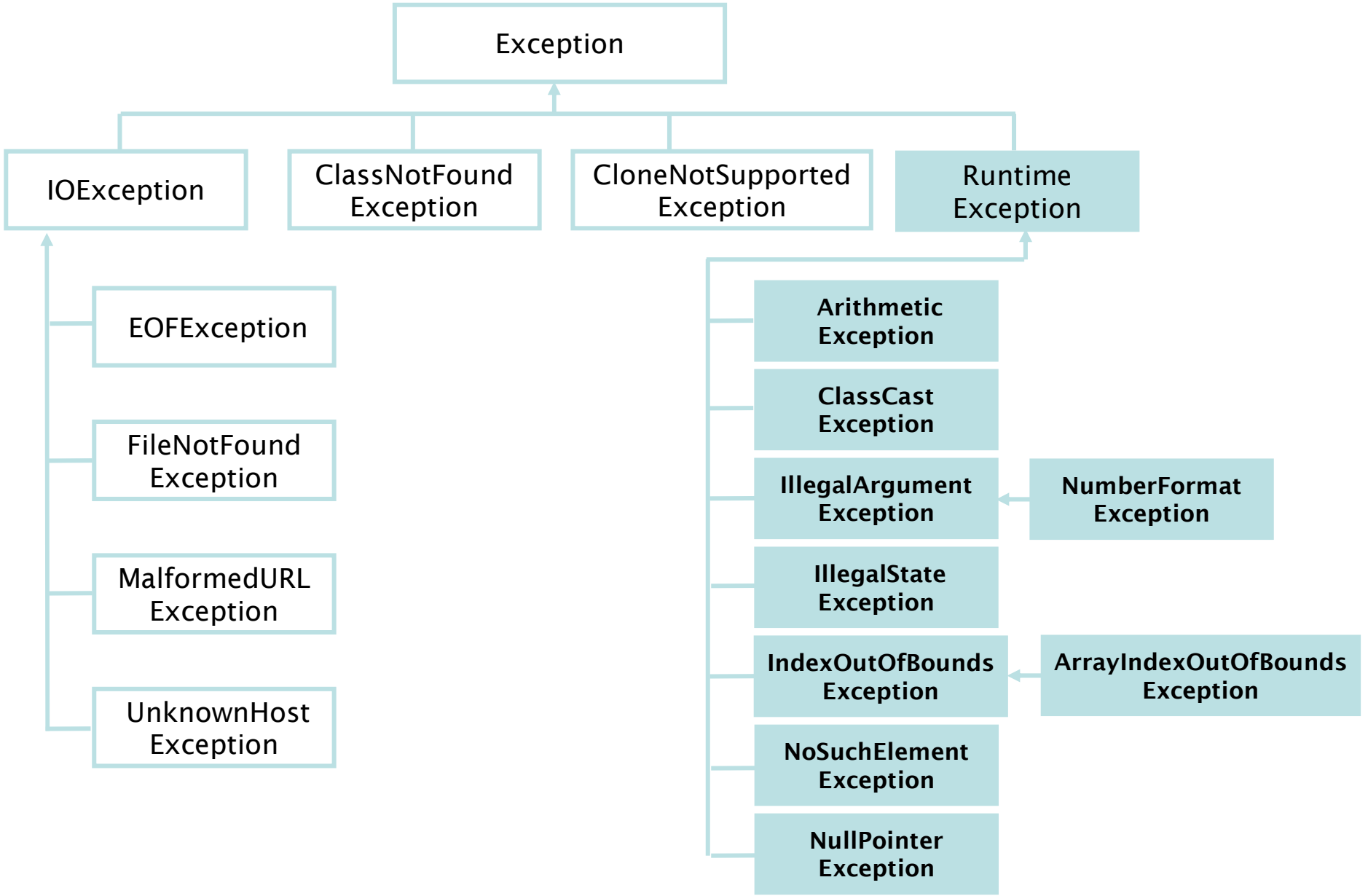
# Tempo di esempi

- Studiare il codice dell'esempio **L01_05**.
  *Questo codice mostra come si possono ottenere risposte diverse scrivendo clausole "catch" diverse.*
  *Per semplificare le cose sono state usate eccezioni "run time" che non richiedono obbligatoriamente il try… catch… ma il meccanismo è eguale anche con le eccezioni "checked".*

# Exception: gerarchia di derivazione da Object



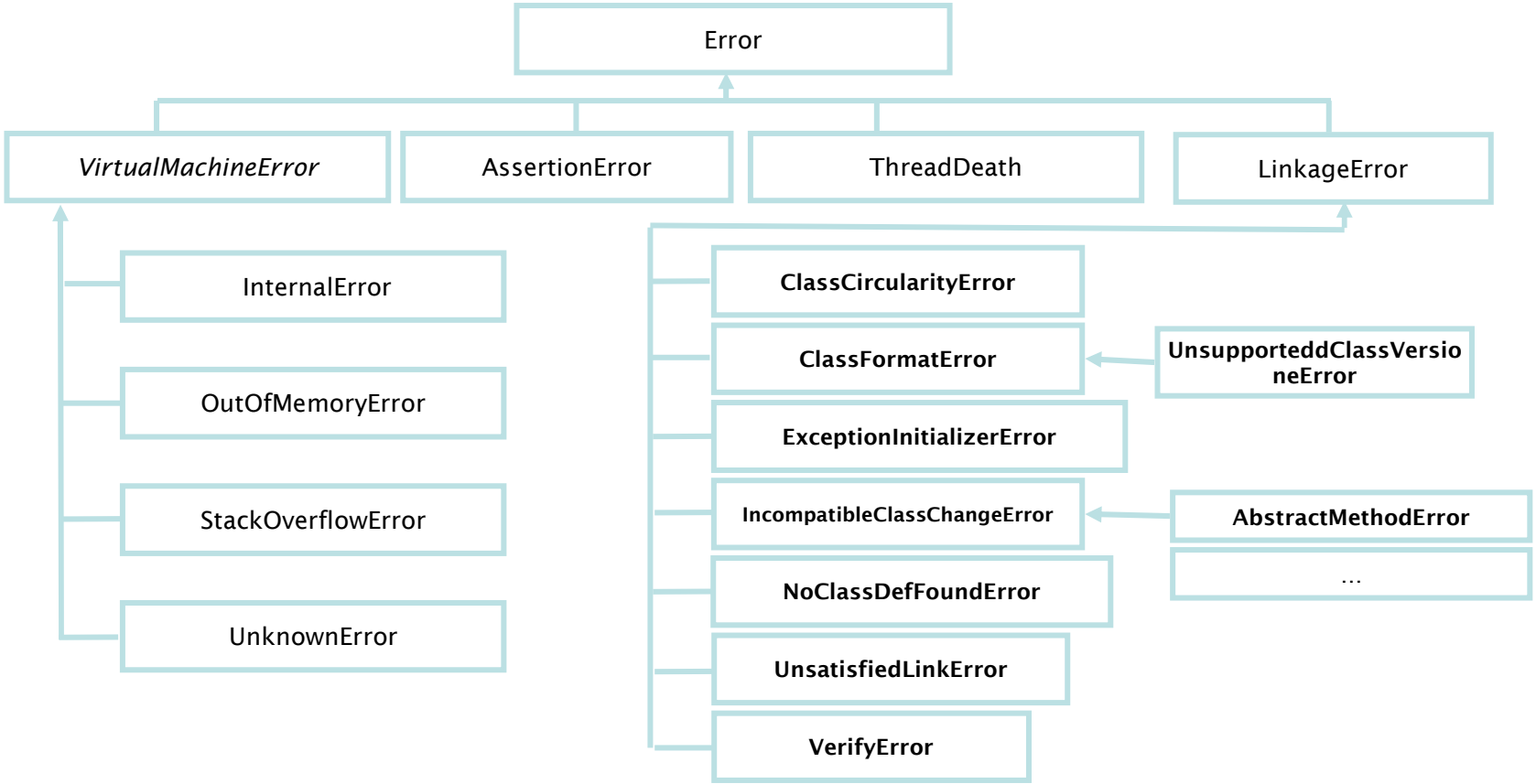
# Gerarchia delle eccezioni predefinite



# Gerarchia degli Errori

Le API Java definiscono anche la classe Error per gli errori  non recuperabili.
Le sottoclassi di Error sono meno comuni di quelle di Exception. Generalmente non bisogna preoccuparsi di tali errori all'interno del codice (e quindi non vanno catturati). Questi indicano errori della virtual machine o errori di sistema.
Quando un errore si verifica:

La JVM visualizza il corrispondente messaggio
quindi causa l'uscita dal programma



# Exception: metodi notevoli

**Costruttori:**

**`Exception()`**: genera una eccezione senza dettagli né “causa”

**`Exception(String detail)`**: genera una eccezione con i dettagli contenuti nella stringa *detail*

**`Exception(Throwable t)`**: genera una eccezione che ha per “causa” t

**`Exception(String detail, Throwable t)`:** genera una eccezione con messaggio “detail” e causa t

**Altri metodi:**

**`toString()`**: restituisce una descrizione generica del messaggio

**`getCause()`**: restituisce la causa

**`getMessage()`** : restituisce il messaggio

# Tempo di esempi

- Studiare il codice dell'esempio **L01_06**.
  *Questo codice mostra l'utilizzo di alcuni dei metodi e costruttori notevoli degli oggetti della famiglia Exception.*

# finally

## *(e prima di terminare… fai comunque questo…)*

Il blocco try … catch… può concludersi con la clausola "finally". Essa specifica delle istruzioni da eseguire SEMPRE sia che le istruzioni dentro il try vadano in porto senza generare eccezioni sia che esse generino eccezioni.

```
try {
      //blocco di istruzioni da provare
    }
catch (ExceptionType ref) {
       //istruzioni per recuperare
     }
finally {
       // comunque vada fai le cose
       // scritte qui
      }
```

Si può usare per "far pulizia" alla fine comunque vadano le cose.

# finally

- Un blocco **`finally`** contiene tipicamente il codice per rilasciare le risorse acquisite nel corrispondente blocco **`try`**. Questa tecnica costituisce un modo efficace per eliminare il problema della perdita di risorse (*resource leak*).
  - Il garbage collector evita i *memory leak.*
- Il blocco **`finally`**
  - Viene inserito dopo l'ultimo blocco catch
  - Viene sempre eseguito
  - Se un'eccezione viene scatenata nel blocco **`finally`** deve essere processata con il relativo **`try/catch`**

# Tempo di esempi

- Studiare il codice dell'esempio **L01_07**.
  *Questo codice mostra l'utilizzo della condizione finally.*

# Passare le eccezioni lungo la catena…

- Se un metodo genera una eccezione essa non deve necessariamente essere gestita nel metodo stesso.
- Spesso conviene “passarla” a chi ha chiamato il metodo perché è lì che si possono prendere le contromisure più efficaci oppure perché è utile “centralizzare” la gestione delle eccezioni in un solo metodo “capofila”.
- In tal caso il metodo costruisce l’eccezione chiamando il metodo costruttore e la “lancia” con l’istruzione “`throw`” verso il metodo che lo ha chiamato.

# Usare il comando `throw`
(scatenare un'eccezione)

- Il comando **`throw`**
  - Scatena un'eccezione quando qualcosa va a mal fine
  - Deve essere seguito da un oggetto eccezione.
    - E' un operatore unario, l'operando è un oggetto di una classe derivata da **Throwable**

```
if ( denominatore == 0 )
   throw new  ArithmeticException();
```

- Quando un'eccezione viene scatenata… ancora una volta
  - Si esce dal blocco corrente e si procede con il corrispondente blocco **`catch`** (se esiste)

# Passare le eccezioni lungo la catena…

- Il metodo segnala al compilatore che potrebbe lanciare eccezioni aggiungendo nel suo header la dichiarazione

  **“throws EccezioneDiQualcheTipo”.**

- Attenzione però: adesso il compilatore ha la possibilità di sapere prima del run-time della presenza di situazioni potenzialmente “eccezionali” e non autorizzerà la compilazione di un codice che non preveda l’exception handler: abbiamo creato una *checked expression*

# Usare la clausola `throws`

- Clausola **`throws`**
  - Specifica le eccezioni che il metodo può scatenare
  - Appare dopo il nome e l'elenco dei parametri del metodo ma prima del corpo
  - Contiene una lista di eccezioni separate da virgola
  - Le eccezioni possono essere scatenate nel corpo del metodo oppure all'interno di metodi chiamati dal metodo stesso
  - Le eccezioni possono essere del tipo specificato nella clausola throws o sue sottoclassi

# Tempo di esempi

- Studiare il codice dell'esempio **L01_08**. *Alcuni metodi si chiamano a catena fino ad uno che "lancia" una eccezione. Se non si aggiunge la clausola "throws" alla intestazione del metodo che lancia la eccezione la compilazione è impossibile.*
- Studiare il codice dell'esempio **L01_09**. *Tutti i metodi che compongono la catena debbono contenere la clausola "throws".*
- Studiare il codice degli esempi **L01_10** **L01_11**. *Qui elaboriamo il precedente esempio in modo che ogni metodo della catena aggiunga qualcosa di suo alla eccezione che porterà traccia del cammino percorso. Questo può essere comodo per operazioni di debugging.*

# Dichiarare nuovi tipi di eccezioni

- Breve richiamo sull'ereditarietà…
  - Per creare una nuova eccezione bisogna estenderne una esistente (Exception o una delle sue sottoclassi).
- Tipicamente contengono solo due costruttori
  - Uno non prende argomenti e passa un messaggio di default al costruttore della superclasse
  - Un riceve un messaggio personalizzato come stringa e lo passa al costruttore della superclasse

# Tempo di esempi

- Studiare il codice dell'esempio **L01_12**.
  *Esempio di creazione di una nuova eccezione.*

# Throwable: metodi notevoli

## `printStackTrace`, `getStackTrace` e `getMessage`

La classe Throwable è la classe di base per gli oggetti che possono essere “thronw” (lanciati) ovvero che possono essere il parametro del comando throw

I metodi nella classe **`Throwable`** danno più informazioni sull’eccezione

**`printStackTrace()`:** produce lo *stack trace* in output sullo standard error

**`getStackTrace()`:** restituisce lo *stack trace* sottoforma di array di oggetti di tipo

**`StackTraceElement`:** consente l’elaborazione personalizzata delle informazioni prodotte dall’eccezione

# Throwable: metodi notevoli
## `printStackTrace`, `getStackTrace` e `getMessage`

- Metodi di **`StackTraceElement`**
  - **`getClassName()`**
  - **`getFileName()`**
  - **`getLineNumber()`**
  - **`getMethodName()`**

- Le informazioni dello stack trace hanno il seguente formato
  *`className.methodName(fileName:lineNumber)`*

# Quando e perché utilizzare tali metodi?
## `printStackTrace`, `getStackTrace` e `getMessage`

- Facciamo un passo indietro…
  - Un'eccezione non gestita provoca una chiamata al gestore di default delle eccezioni Java. Questo comporta la visualizzazione del **nome** dell'eccezione, **la stringa di caratteri facoltativa** fornita quando l'eccezione è stata costruita e lo **stack completo** dell'esecuzione che mostra tutte le chiamate di metodo. Ciò permette al programmatore di vedere il percorso di esecuzione classe dopo classe, metodo dopo metodo, che ha portato a questa eccezione.

**Dall'esempio L01_01**

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at L01_01.metodoConDivisionePerZero(L01_01.java:40)
        at L01_01.metodo2(L01_01.java:35)
        at L01_01.metodo1(L01_01.java:30)
        at L01_01.main(L01_01.java:25)

Process L01_01 exited abnormally with code 1
```

  - In un'eccezione gestita possiamo scegliere cosa stampare. Stampare lo stack trace, con il comando printStackTrace è utile perché percorsi diversi spesso conducono allo stesso metodo. Alcune chiamate al metodo possono generare eccezioni altre no. La stampa dello stack trace risulta di notevole aiuto per individuare la specifica situazione d'errore.

# Tempo di esempi

- Studiare il codice dell'esempio **L01_13**.

  *Viene scatenata un'eccezione e viene stampato lo stackTrace in due modi differenti.*

# Una tentazione da evitare…

La presenza di “unchecked exception” come quelle “run time” ha fatto storcere il naso ad alcuni puristi dei linguaggi di programmazione. La ragione per tale perplessità nasce dalla possibilità di usare malamente la possibilità di generare unchecked expression adottandole in situazioni che invece vanno gestite esplicitamente.

*CATTIVO ESEMPIO: si scrive un metodo che lavora con i giorni dei mesi mettendoli in un array. Il cattivo programmatore scrive i metodi che trattano i mesi con cicli con bounds da 0 a 30. Per evitare i guai posti dai mesi di 28 o 30 giorni egli genera una eccezione di tipo unchecked e scrive exception handler per i casi dei mesi corti. In questo caso le eccezioni sono state usate in modo del tutto improprio!*

Una buona disciplina di programmazione evita di usare le eccezioni per scrivere codice meno accurato!!!

# Malfunzionamento o situazione eccezionale?

Una vecchia barzelletta recita che un buon venditore commerciale di software è sempre in grado di convincere il cliente che i difetti e i “bug” del software che vende sono “undocumented features”.

Da punto di vista pratico però potere prevedere un ragionevole (piccolo) numero di casi in cui il programma segnalerà la presenza di condizioni eccezionali è una scelta corretta e eticamente accettabile.

# Criterio generale

- Se si può (ragionevolmente) sperare di risolvere il problema che fa nascere l'eccezione: scrivere "eccezioni checked". Il senso è: provare a rimediare la situazione perchè si ha speranza di poterlo fare.

- Se non si può fare nulla (ragionevolmente) per risolvere il problema che fa nascere l'eccezione: trattarla come "unchecked" quindi non gestirla. Il senso è: non c'è nulla da fare e quindi si affida la situazione al run time system sperando che almeno lui riesca a fare qualcosa.

# Alla fine:
# a che servono le eccezioni?

- A scrivere codice non offuscato e complicato dalla necessità di dovere tenere conto di casi speciali (tipicamente rari);
- A raccogliere in un solo punto il trattamento di problemi comuni che potrebbero nascere in posti diversi del codice.
- SONO INDISPENSABILI ALLA GESTIONE DEGLI STREAM (input, output, file)…