Programmazione II

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 (mai fidarsi!!!)
- Si raccomanda inoltre la lettura del tutorial ufficiale della SUN a riguardo (file zippato sul sito del corso)

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 la 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.

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 piu'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 principi 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 che durante la normale esecuzione di un programma interrompe il normale flusso di istruzioni.

Supponiamo che durante la 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 al programma che sta simulando la esecuzione della JVM).

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

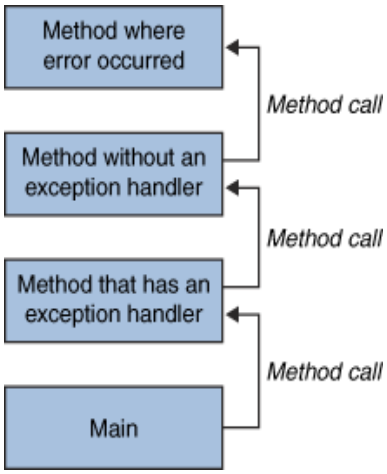
Il *run time system* a sua volta cercherà di trovare qualcosa o qualcuno che possa porre rimedio al problema. Se tale "rimedio" non si trova il programma si arresta, altrimenti viene adottato il rimedio.

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

La “catena di invocazione dei metodi”

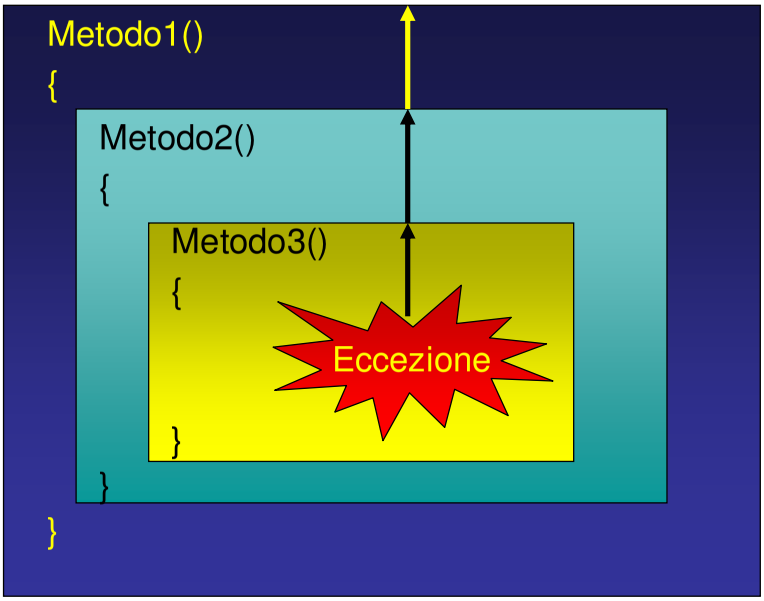
Il codice che deve essere eseguito quando si è verificata l’eccezione, in termine tecnico si chiama “**exception handler**”.

Attenzione, quando viene lanciata l’eccezione il metodo controlla se esso stesso possiede un *exception handler*. Se non è così esso cessa la sua attività e restituisce il controllo del flusso del programma al metodo chiamante, se esso 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**).

Propagazione (*throwing*) di un’eccezione…



La propagazione a ritroso va gestita opportunamente.

Exception handler

L'*exception handler* a seconda dei casi può effettivamente eseguire un *recover* dall'errore 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:

- **Leggibilitá**
- **Manutenibilitá**

Checked vs Unchecked exceptions

Non tutti i problemi che portano alla 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 VIENE BLOCCATA.**

Checked vs Unchecked exceptions

Non tutti i problemi che portano alla 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 la esecuzione del programma. In questo caso si parla di espressioni **"unchecked"** (forse si dovrebbe dire "uncheckable"!).

- **ESEMPI:** dividere per zero in una complessa 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)

La sintassi di questa costruzione è molto semplice:

```
try {
       //blocco di istruzioni
    }
catch (tipoException e)
     {
     //codice da eseguire se
     //qualcosa nel blocco try
     //genera una eccezione del tipo
     //specificato nella clausola"catch"
     }
```

Attenzione: possono esserci più catch, purchè prevedano eccezioni di tipo differente

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 da javac.

Tempo di esempi

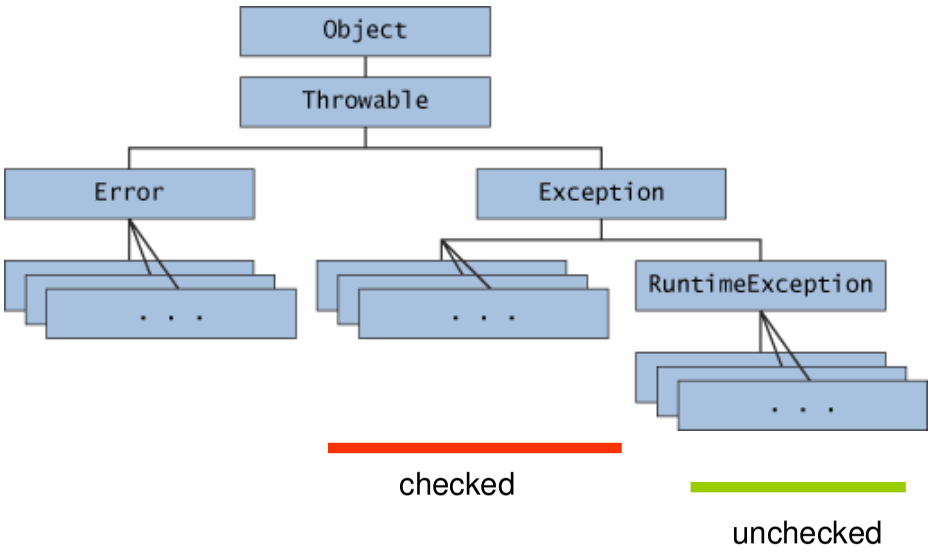
- Studiare il codice dell'esempio **L01_03**.
  Tale codice è lo stesso dell'esempio L01_02 ma con un exception handler (costrutto "try … catch").

- 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!!!

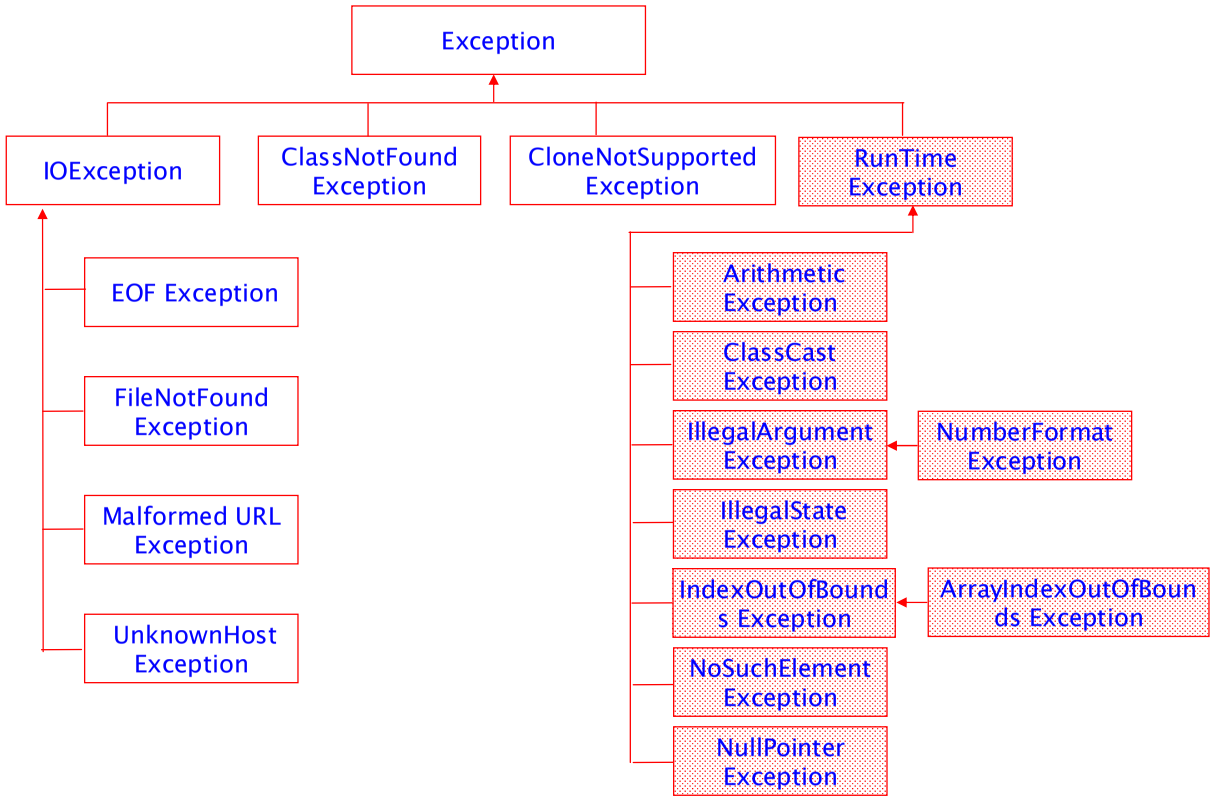
Tempo di esempi

- 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



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 "ammugghiare tutto" fai così…)

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 (tipoException e)
      {
       //istruzioni per recuperare
     }
finally
      {
       // comunque vada fai le cose
       // scritte qui
      }
```

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

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 la istruzione "**throw**" verso il metodo che lo ha chiamato.

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*

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 di suo alla eccezione che porterà traccia del cammino percorso. Questo può essere comodo per operazioni di debugging.*

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!!! Se usate correttamente le eccezioni unchecked invece aggiungono robustezza e stabilità al codice!!

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: scrivere "eccezioni unchecked". 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)…