

# LEZIONI 41-42

## PROGETTARE LE ECCEZIONI

### ... e loro modello in Java

Ingegneria del Software e Progettazione Web  
Università degli Studi di Tor Vergata - Roma

Guglielmo De Angelis  
guglielmo.deangelis@isti.cnr.it

# gestione degli errori – 1

- differenti momenti per differenti tipi di errore:
  - errori di sintassi: compilazione
  - errori di comportamento: fase di test di unità
  - errori di interazione: fase di test di integrazione
- non sempre (i.e. nella pratica quasi mai) è possibile essere certi dell'assenza di errori in un sistema software
  - un sistema in esecuzione può “andare in crash”!!
- spesso è opportuno progettare e sviluppare delle logiche di controllo e gestione degli errori
  - parallelamente alla progettazione dei comportamenti desiderati

# gestione degli errori – 2

- storicamente questo problema è stato gestito mediante la definizione di convenzioni
  - particolari valori di ritorno (e.g. mondo C o Unix-like)
  - flag globali che vengono esaminati prima di consumare risultati di elaborazioni
  - ... ..
- linguaggi di progettazione/programmazione avanzati prevedono costrutti nativi per l'esplicita definizione e gestione del problema:
  - UML State Machine : exit points
  - Java : `java.lang.Exception`

# definizione

- ECCEZIONE: evento che si verifica durante l'esecuzione di una sequenza di azioni in *logica di controllo* e che ne interrompe il flusso così come definito dal progettista/sviluppatore
- per esempio, possono considerarsi eccezioni gli eventi:
  - si rompe l'HD
  - si accede ad un indice fuori dell'array
  - la connessione di rete diviene assente
  - la (de)serializzazione di un oggetto non va a buon fine
  - il file che si vuole accedere è protetto in lettura

# gestione della logica di errore ...

... nel caso UML State Machines

# esempio 1 – 1

**package** Data[  staticView ]

## **FileLoader**

*attributes*

-fileName  
-memArea

*operations*

+setFileName()  
+loadFile()  
+unloadFile()  
-readInMem()  
-releaseMem()

# esempio 1 – 2

**package** Data [  staticView ]


## FileLoader

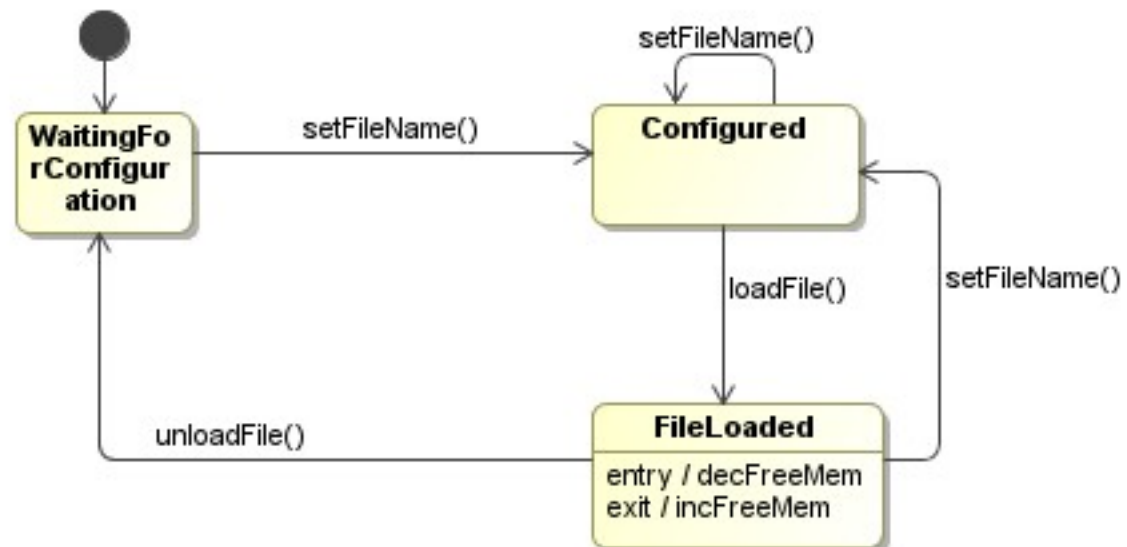
*attributes*

-fileName  
-memArea

*operations*

+setFileName()  
+loadFile()  
+unloadFile()  
-readInMem()  
-releaseMem()

**state machine** FileLoader [  FileLoader ]



# esempio 1 – 3

**package** Data [  staticView ]


## FileLoader

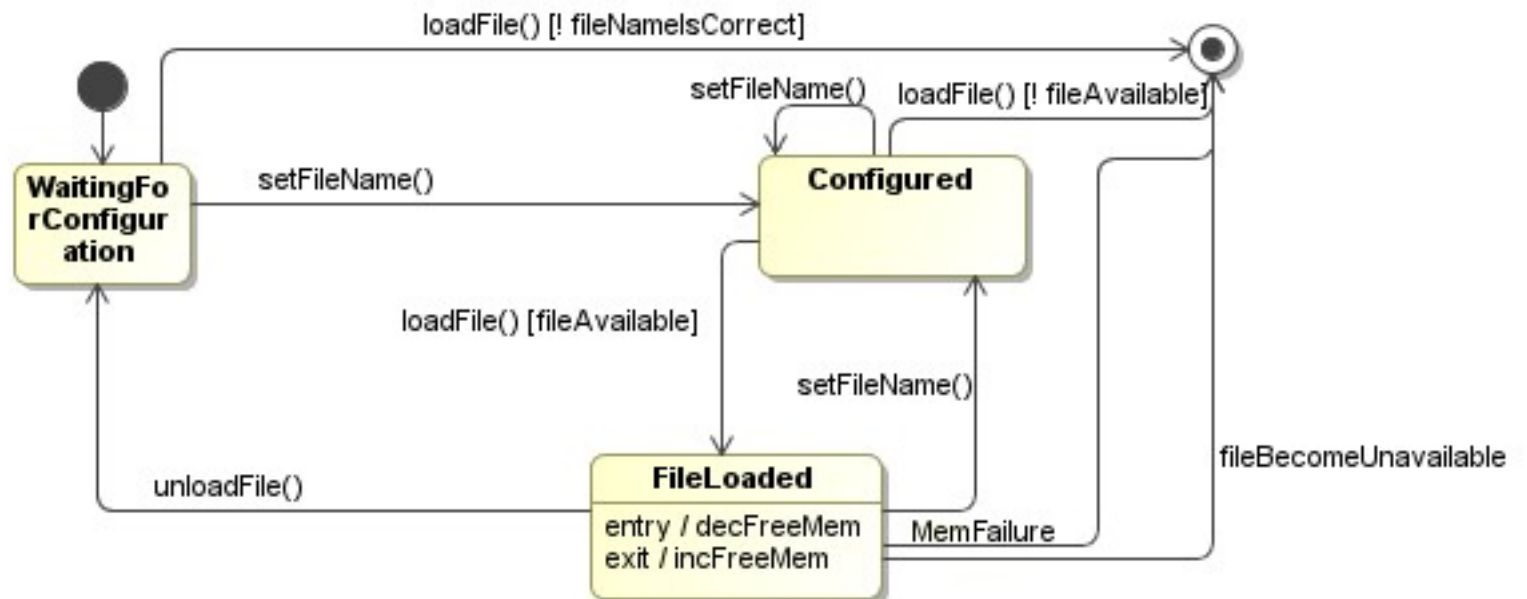
*attributes*

-fileName  
-memArea

*operations*

+setFileName()  
+loadFile()  
+unloadFile()  
-readInMem()  
-releaseMem()

**state machine** FileLoader [  FileLoader ]



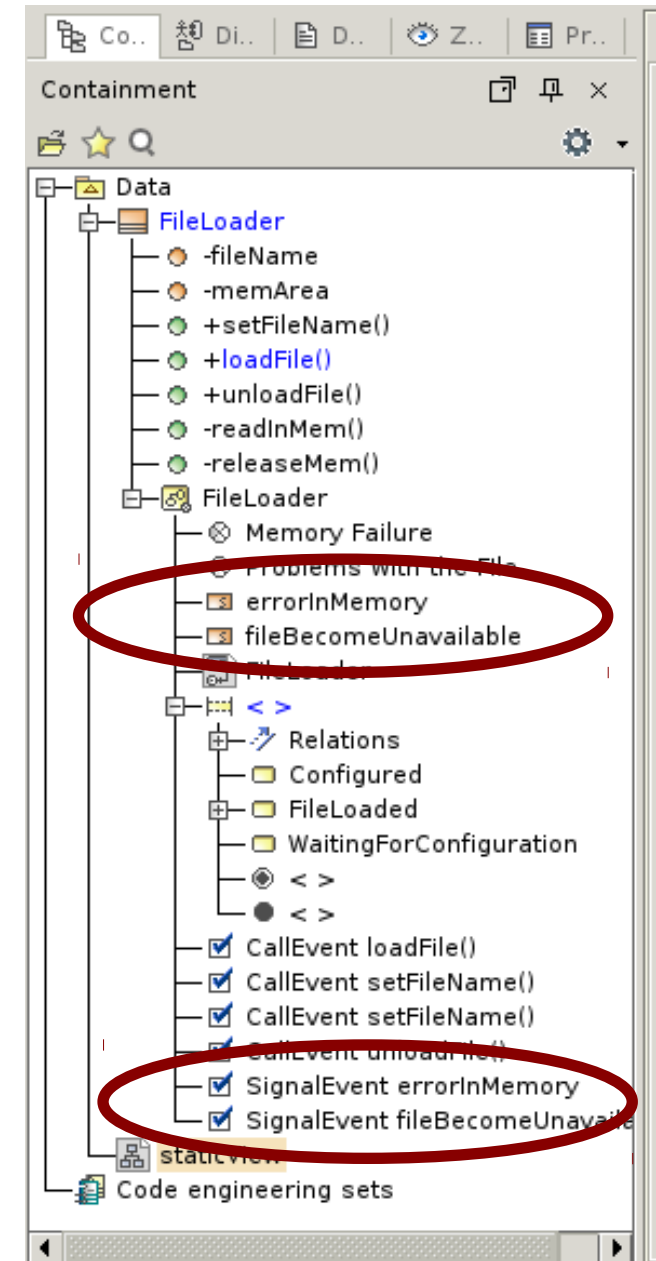


# eccezioni in UML State Machine

- modellare le eccezioni (i.e. i casi di errore) per mezzo di segnali
  - UML Signals
- dare ad ogni segnale una precisa semantica
  - almeno attraverso la descrizione associata e/o il suo nome
- includere nella macchina a stati transizioni che si attivano al ricevimento dei segnali di errore
- gestire i flussi di errore:
  - localmente nella state machine
  - direzionando i flussi che non possono essere gestiti all'esterno, verso opportuni “exit point”
    - uno per tipologia di errore da trattare

# eccezioni in UML State Machine

- modellare le eccezione (i.e. i casi di errore) per mezzo di segnali
  - UML Signals
- dare ad ogni segnale una precisa semantica
  - almeno attraverso la descrizione associata e/o il suo nome
- includere nella macchina a stati transizioni che si attivano al ricevimento dei segnali di errore
- gestire i flussi di errore:
  - localmente nella state machine
  - direzionando i flussi che non possono essere gestiti all'esterno, verso opportuni “exit point”
    - uno per tipologia di errore da trattare



# esempio 1 – 4

**package** Data[  staticView ]


## FileLoader

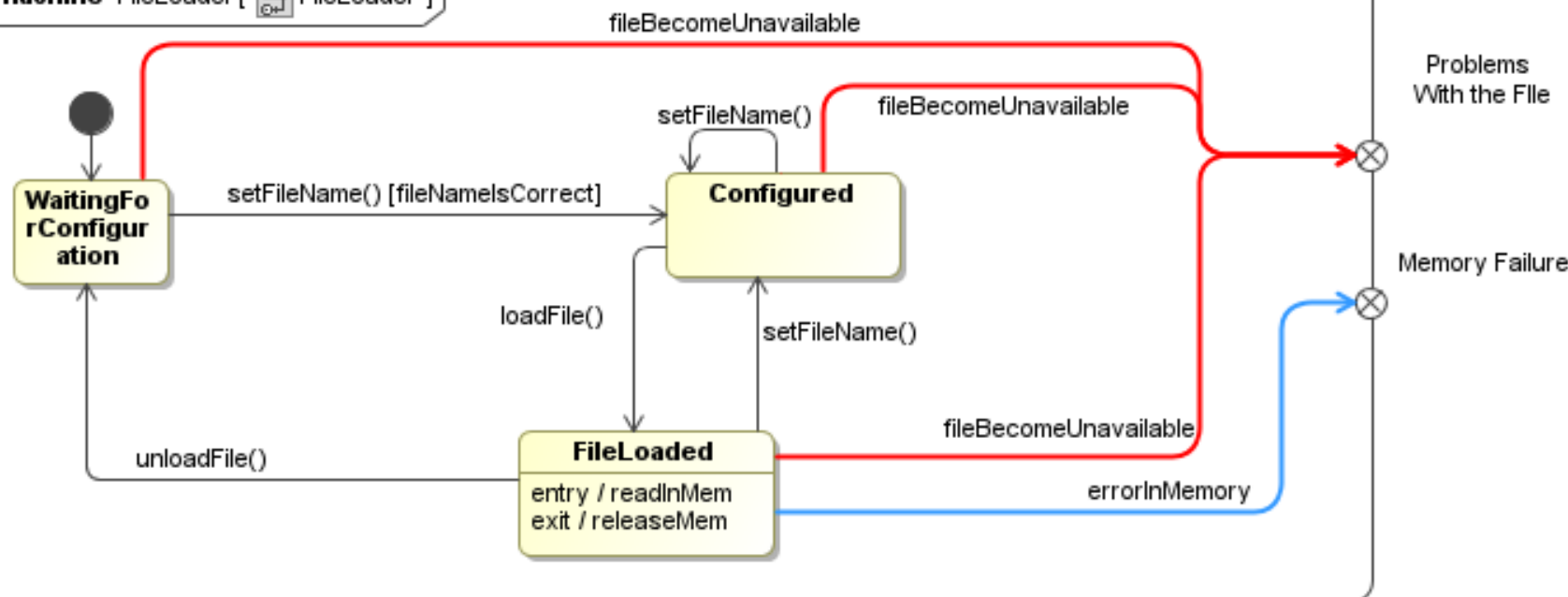
*attributes*

-fileName  
-memArea

*operations*

+setFileName()  
+loadFile()  
+unloadFile()  
-readInMem()  
-releaseMem

**state machine** FileLoader [  FileLoader ]



# gestione della logica di errore ...

... nel caso UML State Machines

riassunto :

1. modellare un segnale per ogni tipologia di errore
2. definire un evento per ogni segnale considerato
3. dipendentemente dalla politica di gestione e recupero dell'errore
  - definire opportune transizioni attivabili attraverso gli eventi di errore considerati
  - identificare uno o più "exit point" nella State Machine

# gestione della logica di errore ...

## ... nel caso Java

## esempio 2 – 1

```
readFile {  
    open(file);  
    s = size(file);  
    mem = allocate(s);  
    read(file, mem);  
    close(file);  
}
```

## esempio 2 – 2

- scrivere una funzione che apre un file, ne verifica la dimensione, e lo copia in memoria dopo averne allocato sufficiente spazio

```
readFile {  
    open(file) ;  
    s = size(file) ;  
    mem = allocate(s) ;  
    read(file, mem) ;  
    close(file) ;  
}
```

## esempio 2 – 3

- scrivere una funzione che apre un file di determinata dimensione, e lo copia in memoria, assicurandosi di allocare sufficiente spazio

```
readFile {  
    open(file);  
    s = size(file);  
    mem = allocate(s);  
    read(file, mem);  
    close(file);  
}
```

- sembra tutto ok ma :
  - se il file non può essere aperto?
  - se la lunghezza del file non può essere determinata?
  - se non vi è abbastanza memoria da allocare?
  - se la lettura fallisce?
  - se il file non può essere chiuso?



## esempio 2 – 4

```
readFile {
    errcode = 0;
    open(file);
    if (!failed){
        s = size(file);
        if ( s>=0 ){
            mem = allocate(s);
            if (isFree(mem)){
                read(file, mem);
                if (!failed)
                    errcode = -1;
            }else
                errcode = -2;
        }else
            errcode = -3;

        close(file);
        if (closeFailed && errorCode == 0)
            errorCode = -4;
        else
            if (errorCode != 0)
                errorCode = -5;
        }else
            errorCode = -6;

    return errorCode;
}
```

- adesso sapete dire che cosa fa questo programma “ad occhio” ?
  - probabilmente no: logica di controllo e logica di errore mischiate
- 5 statement iniziali VS. 23 attuali
  - +360%
- cosa accadrebbe se dovessi modificare questo codice o una codifica di errore?
  - maggiore difficoltà nella manutenzione

## esempio 2 – 5

```
readFile {  
    try{  
        open(file);  
        s = size(file);  
        mem = allocate(s);  
        read(file, mem);  
        close(file);  
    }catch (failedOpen) {  
        ... ..  
    }catch (noSize) {  
        ... ..  
    }catch (failedCopy) {  
        ... ..  
    }catch (failedClose) {  
        ... ..  
    }  
}
```

- soluzione elegante e compatta dove le logiche di controllo e quella di errore sono progettate e gestite separatamente

# try/catch Java – 1

- si tenta di eseguire il codice e se si intercetta un'eccezione vi pone rimedio

- sintassi

```
try {  
    <IstruzioniLogicaDiControllo>  
} catch (<Tipo eccezione> <Identificatore>) {  
    <IstruzioniLogicaErrore>  
}
```

- il costrutto `try` definisce un blocco di istruzioni in cui può verificarsi un'eccezione
- un blocco `try` è seguito da una o più clausole `catch`, che specificano quali eccezioni vengono gestite
  - ogni clausola corrisponde a un tipo di eccezione sollevata
  - al verificarsi di un'eccezione, la computazione “salta” alla prima istruzione della prima clausola che corrisponde all'eccezione sollevata
  - il match avviene in base al tipo (catena di generalizzazioni)
  - la gestione dell'eccezione è soddisfatta al termine della clausola `catch` trovata

# try/catch Java – 2

- il blocco `try/catch` può avere una clausola `finally` opzionale
- schema di funzionamento
  - non viene sollevata nessuna eccezione
    - le istruzioni nella clausola `finally` vengono eseguite dopo che si è concluso il blocco `try`
  - si verifica un'eccezione catturata dai comparti `catch`
    - le istruzioni nella clausola `finally` vengono eseguite dopo le istruzioni della clausola `catch` appropriata
  - si verifica un'eccezione **NON** prevista dai comparti `catch`
    - le istruzioni nella clausola `finally` vengono eseguite prima dell'inoltro dell'eccezione al chiamante del metodo
- SE presente clausola `finally` viene **SEMPRE** eseguita

??? ??? ???

# try/catch Java – 2

- il blocco `try/catch` può avere una clausola `finally` opzionale
- schema di funzionamento
  - non viene sollevata nessuna eccezione
    - le istruzioni nella clausola `finally` vengono eseguite dopo che si è concluso il blocco `try`
  - si verifica un'eccezione catturata dai comparti `catch`
    - le istruzioni nella clausola `finally` vengono eseguite dopo le istruzioni della clausola `catch` appropriata
  - si verifica un'eccezione **NON** prevista dai comparti `catch`
    - le istruzioni nella clausola `finally` vengono eseguite prima dell'inoltro dell'eccezione al chiamante del metodo
- SE presente clausola `finally` viene **SEMPRE** eseguita
  - utilizzato per liberare risorse utilizzate all'interno del blocco `try/catch` (es. Files, DB)
  - l'unico caso in cui non viene eseguito è a seguito dell'invocazione di `“System.exit(int n)”`

# esempi eccezioni Java – 1

VEDERE MATERIALE ALLEGATO  
ALLA LEZIONE :

`TryCatchFinallyClass.java`

`(howFinallyWorks, thisIsTheOnlyCaseFinallyIsSkipped)`

# throws Java – 1

- le eccezioni possono essere racchiuse all'interno di un blocco `try/catch`
  - il blocco definisce i criteri di gestione dell'eccezione
- ... e se non si è in grado di stabilire opportunamente la gestione di un errore?!?!?!?

# throws Java – 1

- le eccezioni possono essere racchiuse all'interno di un blocco `try/catch`
  - il blocco definisce i criteri di gestione dell'eccezione
- ... e se non si è in grado di stabilire opportunamente la gestione di un errore?!?!?!?
  - è necessario identificare il metodo come possibile sorgente di “errore non gestito”
  - il chiamante del metodo deve farsi carico di gestire l'eventuale eccezione sollevata e non gestita



# throws Java – 2

- la dichiarazione del metodo che può sollevare una eccezione è marcata con la clausola `throws`
- è ammesso che un metodo catturi eccezioni e poi le rilanci mediante clausola `throw`
- in ogni caso il chiamante che deve racchiudere l'invocazione di tale metodo all'interno di un blocco `try/catch`

# esempi eccezioni Java – 2

VEDERE MATERIALE ALLEGATO  
ALLA LEZIONE :

`TryCatchFinallyClass.java`  
(howThrowWorks)

# progettare le eccezioni

- progettare la logica di errore è uno step fondamentale della progettazione di un sistema software
- anche per le eccezioni è possibile applicare le tecniche/strategie/pattern visti fino a questo momento
- nella visione O.O., tutto è un oggetto
- anche le eccezioni sono visti come oggetti
  - con tutte le caratteristiche che ne conseguono

# gerarchia di eccezioni

- consente di definire una tassonomia degli errori cui il sistema può incorrere
- semplifica la definizione di strategie di recovery o di segnalazione degli errori in corso
  - i.e. la progettazione della logica di errore
- semplifica la manutenzione della logica di errore

# gerarchia di eccezioni – UML

- consente di definire una tassonomia degli errori cui il sistema può incorrere
  - semplifica la definizione di strategie di recovery o di segnalazione degli errori in corso
    - i.e. la progettazione della logica di errore
  - semplifica la manutenzione della logica di errore
- esplicita la definizione di un insieme di segnali opportunamente organizzati mediante la relazione di generalizzazione

# gerarchie di eccezioni – Java

- un oggetto eccezione è sempre un'istanza di una classe derivata da `java.lang.Throwable`
- gerarchia di base suddivisa in due macro categorie
  - errori : `java.lang.Error`
  - eccezioni : `java.lang.Exception`

# gerarchie di eccezioni – Java

errori:

- anomalie che si verificano all'interno della VM (e.g. dynamic linking, hard failure, ect)
  - `OutOfMemoryError`
  - `StackOverflowError`
- difficilmente è possibile recuperare da errori di questo tipo

# esempi eccezioni Java – 3

VEDERE MATERIALE ALLEGATO  
ALLA LEZIONE :

`TriggerStackOverflow.java`

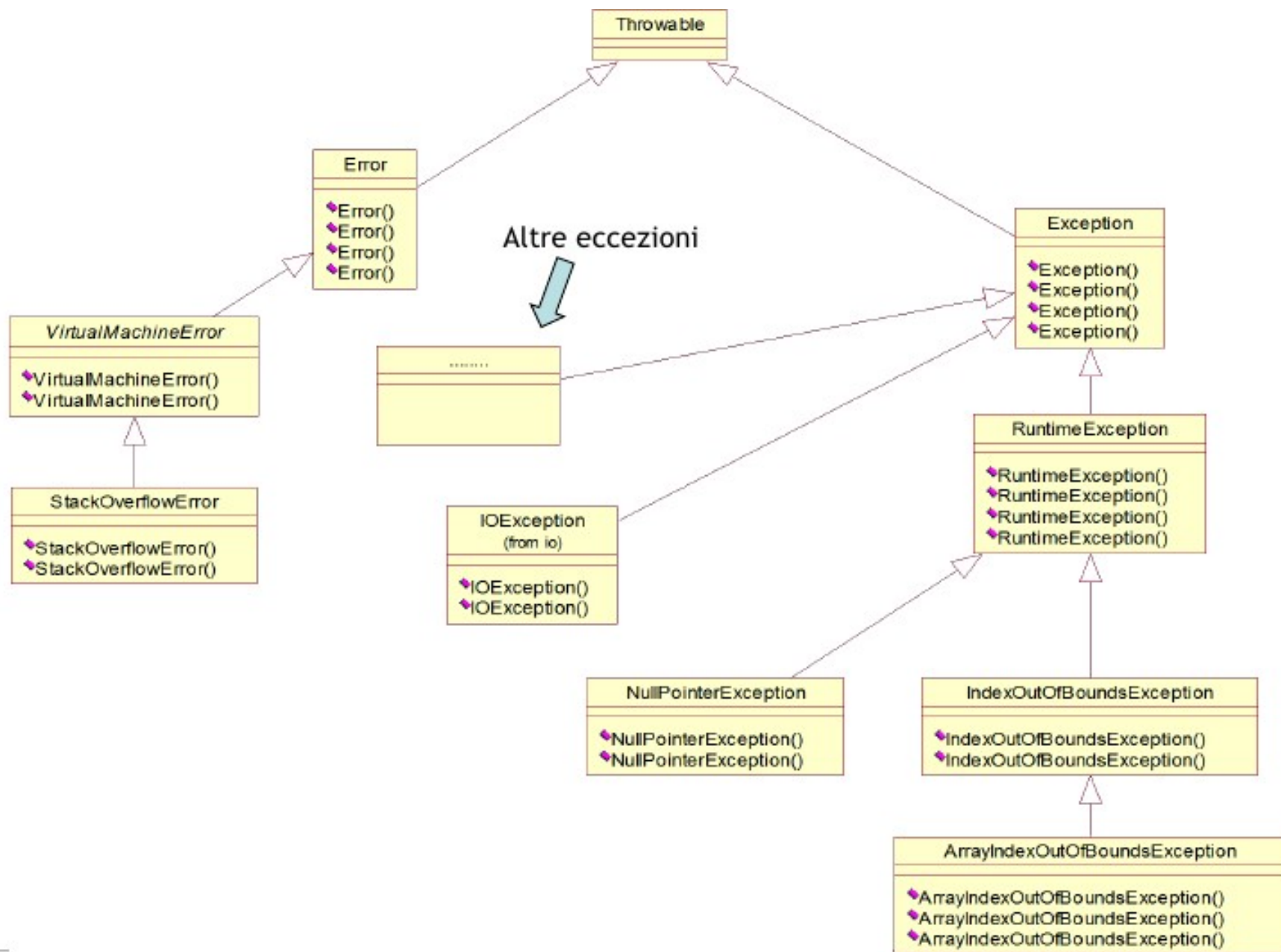


# gerarchie di eccezioni – Java

eccezioni :

- `java.lang.RuntimeException` e sue sottoclassi
  - sono dette unchecked (non verificate)
  - non è obbligatorio gestirle
  - si verificano quando è stato commesso un errore di programmazione
    - cast definito male: `ClassCastException`
    - accesso ad un puntatore nullo: `NullPointerException`
- altre classi che **NON** derivano da `RuntimeException`
  - sono dette checked (verificate)
  - è obbligatorio gestirle in un blocco `try/catch` oppure delegarne la gestione usando la clausola `throws` nel metodo che le genera
  - si verificano quando si è verificato qualcosa di imprevisto
    - apertura di un file: `FileNotFoundException`

# gerarchie di eccezioni – Java



# operazioni in Throwable

- `getMessage()`
  - accede al messaggio contenente i dettagli dell'eccezione
- `toString()`
  - restituisce una breve descrizione dell'oggetto Throwable, compresi i dettagli dell'eccezione, se esistenti
- `printStackTrace()`
  - stampa sullo standard di error l'eccezione con il relativo stack delle chiamate
- `getCause()`
  - da Java 1.4 è possibile impostare l'eccezione originale come "causa" della nuova eccezione ovvero si possono annidare eccezioni
- `initCause(Throwable cause)`
  - configura/sovrascrive la causa di una eccezione

# esempi eccezioni Java – 4

VEDERE MATERIALE ALLEGATO  
ALLA LEZIONE :

`TestThrowsKeyword.java`

# ... ancora su throws Java

- abbiamo detto che (vedi slide 25)
  - un metodo che può lanciare una eccezione è marcato con la clausola `throws`
  - l'effettivo lancio di una istanza di eccezione avviene per mezzo del comando `throw`
- un metodo ha la possibilità di catturare e rigenerare un'eccezione diversa da quella intercettata
  - conversione esplicita di eccezioni
  - exception chaining/wrapping

# esempi eccezioni Java – 4

VEDERE MATERIALE ALLEGATO  
ALLA LEZIONE :

`ExampleOnExceptionChainingClass.java`

# overriding && eccezioni

- in caso di override, le eccezioni (di tipo "checked") previste dalla specifica di una operazione hanno un impatto anche sull'implementazione dei relativi metodi
- in caso di override di un metodo: nella sottoclasse generare soltanto le eccezioni che possono essere lanciate dal metodo della classe base
  - assicura che il codice che funzioni con la classe base
  - automaticamente funziona con qualsiasi oggetto derivato dalla classe base
  - non può lanciare un'eccezione più generale di quella del metodo della superclasse
- lo stesso principio si applica nella definizione di metodi che implementano operazioni di una interfaccia
- per i costruttori non vale questo principio
  - i costruttori delle sottoclassi possono sollevare qualsiasi eccezione
  - tutte le eccezioni lanciabili nella superclasse devono essere esplicitate anche nel costruttore della sottoclasse
    - il costruttore di una sottoclasse invoca (esplicitamente o implicitamente) il costruttore della superclasse

# esempi eccezioni Java – 5

VEDERE MATERIALE ALLEGATO  
ALLA LEZIONE :

`it.uniroma2.dicii.ispw.exceptionsExamples.overriding1`

**&&**

`it.uniroma2.dicii.ispw.exceptionsExamples.overriding2`