Metodologie di Programmazione

Lezione 21: Le eccezioni

Lezione 21: Sommario

UNITELMA SAPIENZA

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

- Le eccezioni
- Vantaggi e svantaggi
- Catturare le eccezioni
- La politica catch or declare
- Personalizzare le eccezioni
- Esercizi

```
    for ( ... )
        for ( ... ) {
            ...
            if (disaster)
                goto error;
        }
    ...

error:
    clean up the mess
```

Le eccezioni in breve

- Le eccezioni rappresentano un meccanismo utile a notificare e gestire gli errori
- Un'eccezione indica che durante l'esecuzione si è verificato un errore
- Il termine "eccezione" indica un comportamento anomalo, che si discosta dalla normale esecuzione
- Codice più robusto e sicuro

Lettura di un array in C

- Ricordate come si scorre un array in

Ma la dimensione dell'array è pari a 5

```
int estrazioneLotto[5] = { 3, 29, 10, 23, 67 };

for (int i=0; i<=5; i++) printf("%d ", estrazioneLotto[i]);
```

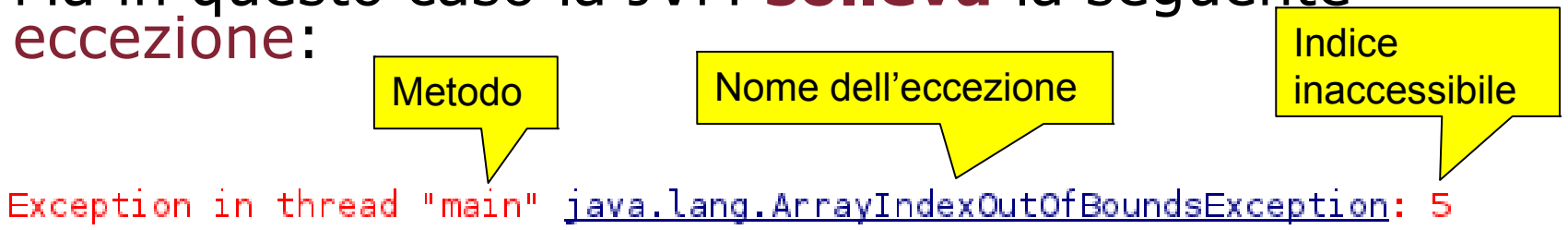
- In questo caso viene acceduta la **sesta** cella del vettore, adiacente all'ultima
- Né programmatore né utente hanno possibilità di accorgersi dell'errore e il programma ha due effetti indesiderati:
  - Lettura di zone di memoria "incontrollate"
  - Spesso una mancata notifica "ad alto livello" del comportamento anomalo

Lettura di un array in Java

- In Java il codice è molto simile:

```
int[] estrazioneLotto = { 3, 29, 10, 23, 67 };

for (int i=0; i<=5; i++) System.out.print(estrazioneLotto[i] + " ");
```

- Ma in questo caso la JVM **solleva** la seguente eccezione:



- L'esecuzione viene interrotta e ci **accorgiamo** del superamento incontrollato dei confini dell'array

Convertire da stringa a intero

- Mediante il metodo statico Integer.parseInt
- Ma se la stringa in input non contenesse un intero:

```
String x = "abc";
int k = Integer.parseInt(x);
```

quale intero dovrebbe restituire il metodo?

- Il metodo emette un'eccezione:

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "x"
```

- L'esecuzione viene interrotta e veniamo informati dell'impossibilità di conversione

Dividere per zero

- E se dividiamo per 0?

```
int x = 5;
int y = 0;

int z = x/y;
```

- L'operatore / emette un'eccezione:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

- L'esecuzione viene interrotta e veniamo informati dell'impossibilità di dividere per zero

Eccezioni notevoli

Eccezione	Descrizione
IndexOutOfBoundsException	Accesso ad una posizione non valida di un array o una stringa (<0 o maggiore della sua dimensione)

ClassCastException	Cast illecito di un oggetto ad una sottoclasse a cui non appartiene.
	Esempio:
	`Object x = new Integer(0);`
	`System.out.println((String)x);`

ArithmeticException	Condizione aritmetica non valida (es. divisione per zero)
CloneNotSupportedException	Metodo **clone**() non implementato o errore durante la copia dell'oggetto
ParseException	Errore inaspettato durante il parsing
IOError e IOException	Grave errore di input o output
IllegalArgumentException	Parametro illegale come input di un metodo
NumberFormatException	Errore nel formato di un numero (estende la precedente)

Perché non restituire un “valore d’errore”?

- Perché bisognerebbe restituire un valore “speciale” per ogni tipo d’errore e prevedere una codifica dei valori d’errore comune a tutti i metodi
- Perché bisognerebbe gestire gli errori **per ogni istruzione eseguita**

Pseudocodice C

```
Svolgi compito 1
Se il precedente compito non è andato a buon fine
Allora controlla e gestisci gli errori

Svolgi compito 2
Se il precedente compito non è andato a buon fine
Allora controlla e gestisci gli errori

Svolgi compito 3
Se il precedente compito non è andato a buon fine
Allora controlla e gestisci gli errori
```

Pseudocodice Java

```
try
{
  Svolgi compito 1
  Svolgi compito 2
  Svolgi compito 3
  Svolgi compito 4
}
catch(ExceptionType1 e1){}
catch(ExceptionType2 e2){}
catch(ExceptionType3 e3){}
catch(ExceptionType4 e4){}
finally{}
```

Le eccezioni: i vantaggi

- In linguaggi come il C, la logica del programma e la logica di gestione degli errori sono interlacciate: questo rende più difficile leggere, modificare e mantenere il codice

- Gli errori vengono propagati verso l'alto lungo lo stack di chiamate

- Codice più robusto: non dobbiamo controllare esaustivamente tutti i possibili tipi di errore: il polimorfismo lo fa per noi, scegliendo l'intervento più opportuno

Le eccezioni: gli svantaggi

- L'onere di gestire i vari tipi di errore si sposta sulla JVM che si incarica di capire il modo più opportuno per gestire la situazione di errore

Che cosa si può gestire con le eccezioni

- Errori sincroni, che si verificano a seguito dell'esecuzione di un'istruzione
  - **Errori non critici**: errori che derivano da condizioni anomale
    - divisione per zero
    - errore di I/O
    - errori durante il parsing
  - **Errori critici o irrecuperabili**: errori interni alla JVM
    - conversione di tipo non consentito
    - accesso ad una variabile riferimento con valore null
    - mancanza di memoria libera
    - riferimento a una classe inesistente

Che cosa NON si può gestire con le eccezioni

- Eventi asincroni
  - completamenti nel trasferimento I/O
  - ricezione messaggi su rete
  - click del mouse
- Eventi che accadono **parallelamente** all'esecuzione e quindi **indipendenti** dal flusso di controllo

Catturiamo le eccezioni: il blocco try-catch

- Il blocco try-catch consente di **catturare** le eccezioni

Il blocco try

- Nel blocco **try** si inseriscono tutte le istruzioni dalle quali vengono sollevate le eccezioni che vogliamo catturare

```
public class Spogliatoio
{
    public static void main(String[] args)
    {
        Sportivo pellegrini = new Sportivo("Federica Pellegrini");
        Sportivo bolt = new Sportivo("Usain Bolt");

        Armadietto armadietto = new Armadietto(pellegrini);
        try
        {
            armadietto.apriArmadietto(bolt);
        }
        catch (NonToccareLaMiaRobaException e)
        {
            // Provvedimenti contro i ladri...
        }
        catch (ArmadiettoGiaApertoException e)
        {
            // Notifica che l'armadietto e' gia' aperto...
        }
    }
}
```

blocco try

Il blocco catch (1)

- All'interno del blocco **catch** è necessario indicare il tipo di eccezione da catturare e specificare nel blocco le azioni da attuare a seguito dell'eccezione sollevata

```
public class Spogliatoio
{
    public static void main(String[] args)
    {
        Sportivo pellegrini = new Sportivo("Federica Pellegrini");
        Sportivo bolt = new Sportivo("Usain Bolt");

        Armadietto armadietto = new Armadietto(pellegrini);
        try
        {
            armadietto.apriArmadietto(bolt);
        }
        catch (NonToccareLaMiaRobaException e)
        {
            // Provvedimenti contro i ladri...
        }
        catch (ArmadiettoGiaApertoException e)
        {
            // Notifica che l'armadietto e' gia' aperto...
        }
    }
}
```

blocco catch

Il blocco catch (2)

- E' infine possibile specificare molteplici blocchi catch, in risposta a differenti eccezioni sollevate:

```
public class Spogliatoio
{
    public static void main(String[] args)
    {
        Sportivo pellegrini = new Sportivo("Federica Pellegrini");
        Sportivo bolt = new Sportivo("Usain Bolt");

        Armadietto armadietto = new Armadietto(pellegrini);
        try
        {
            armadietto.apriArmadietto(bolt);
        }
        catch (NonToccareLaMiaRobaException e)
        {
            // Provvedimenti contro i ladri...
        }
        catch (ArmadiettoGiaApertoException e)
        {
            // Notifica che l'armadietto e' gia' aperto...
        }
    }
}
```

Possiamo trattare ciascuna eccezione in modo specifico

blocchi catch

Il blocco catch: l'ordine conta!

- E' molto importante considerare l'ordine con cui si scrivono i diversi blocchi catch e catturare le eccezioni dalla più specifica a quella più generale

- Nell'attuare il processo di cattura, la JVM sceglie il primo catch compatibile, tale cioè che il tipo dell'eccezione dichiarata sia lo stesso o un supertipo dell'eccezione lanciata durante l'esecuzione

- Spesso vogliamo rispondere ad un'eccezione con il rimedio specifico e non con uno più generale

Il blocco catch: l'ordine conta!

UNITELMA SAPIENZA — SAPIENZA UNIVERSITÀ DI ROMA, DIPARTIMENTO DI INFORMATICA

```
public class TavoloDiGioco
{
    public enum Seme
    {
        SPADE,
        DENARI,
        BASTONI,
        COPPE;
    }
    public static void main(String[] args)
    {
        try
        {
            Integer due = Integer.parseInt("Due");
            Seme denari = Seme.valueOf("DENARA");

            Carta dueDiDenari = new Carta(due, denari);
        }
        catch(IllegalArgumentException e)
        {
            System.out.println("Seme non esistente...");
        }
        catch(NumberFormatException e)
        {
            System.out.println("Valore non esistente...");
        }
    }
}
```

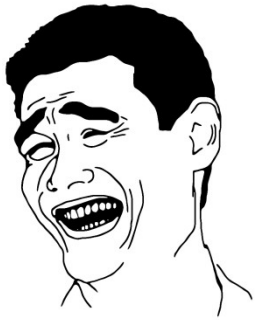
- NumberFormatExcetption estende IllegalArgumentException
- La prima istruzione nel blocco try solleva l'eccezione più specifica, la seconda quella più generale
- Quando viene eseguita la prima istruzione, viene sollevata una NumberFormatException
- Ma nell'analisi la JVM considera il primo blocco catch come compatibile
- Il secondo caso non viene invece mai raggiunto

Flusso in presenza o assenza di eccezioni

- Se durante l'esecuzione non vengono sollevate eccezioni:
  - Ciascuna istruzione all'interno del blocco try viene eseguita normalmente
  - Terminato il blocco try, l'esecuzione riprende dalla prima linea dopo il blocco try-catch
- Se viene sollevata un'eccezione:
  - L'esecuzione del blocco try viene interrotta
  - Il controllo passa al primo blocco catch compatibile, tale cioè che il tipo dichiarato nella clausola catch sia dello stesso tipo dell'eccezione sollevata, o un suo super-tipo
  - L'esecuzione riprende dalla prima linea dopo il blocco try-catch

La politica catch-or-declare (1)

- Una volta sollevata un'eccezione, possiamo:
  - **Ignorare** l'eccezione e propagarla al metodo chiamante, a patto di aggiungere all'intestazione del metodo la clausola throws, seguìto dall'elenco delle eccezioni potenzialmente sollevate (**declare**)
  - **Catturare** l'eccezione, ovvero gestire la situazione anomala in modo opportuno, prendendo provvedimenti e contromisure atte ad arginare il più possibile la situazione di emergenza (**catch**)



La politica catch-or-declare (2)

- Se il requisito catch-or-declare non viene soddisfatto il compilatore emette un errore che indica che l'eccezione dev'essere **catturata** o **dichiarata**
- Questo serve a **forzare il programmatore** a considerare i problemi legati all'uso di metodi che emettono eccezioni

Ignorare le eccezioni (1)

- Se intendiamo ignorare l'eccezione siamo costretti a dichiarare esplicitamente il suo sollevamento con **throws**:

```
public class Spogliatoio
{
    public static void main(String[] args) throws NonToccareLaMiaRobaException, ArmadiettoGiaApertoException
    {
        Sportivo pellegrini = new Sportivo("Federica Pellegrini");
        Sportivo bolt = new Sportivo("Usain Bolt");

        Armadietto armadietto = new Armadietto(pellegrini);
        armadietto.apriArmadietto(bolt);
    }
}
```

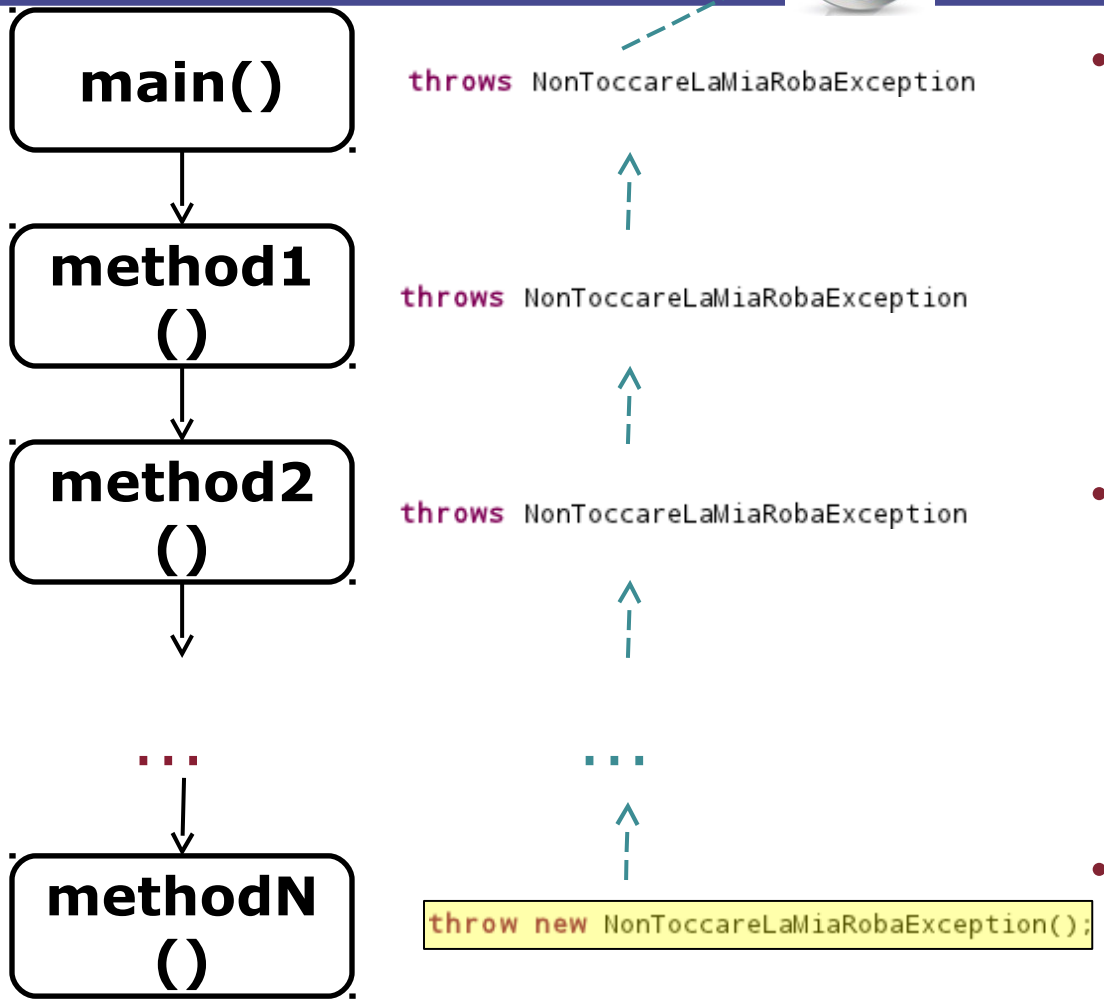
- Il costrutto **throws dichiara** che il metodo (o i metodi delle classi da questo invocati) **può** sollevare eccezioni dello stesso tipo (o di un tipo più specifico) di quelle elencate dopo throws
  - Tale specifica non è sempre obbligatoria, ma dipende dal tipo di eccezione sollevata

Ignorare le eccezioni (2)

- Se **tutti i metodi** all'interno dell'albero delle chiamate dell'esecuzione corrente decidono di ignorare l'eccezione, l'esecuzione viene **interrotta**
  - Questo è vero nel caso di un'applicazione con un singolo thread
  - Nel caso di molteplici thread, è il singolo thread ad essere interrotto; l'applicazione termina se lo sono tutti i thread

Ignorare le eccezioni (3)

```
Exception in thread "main" NonToccareLaMiaRobaException
        at Armadietto.apriArmadietto(Armadietto.java:11)
        at Spogliatoio.main(Spogliatoio.java:10)
```



- Il metodo più in basso nello stack di attivazione lancia un'eccezione (throw point)
- Tutti i metodi decidono di ignorare l'eccezione con throws
- Come effetto si ha la **terminazione** dell'esecuzione

I metodi printStackTrace() e getMessage()

- Quando un'eccezione non viene mai catturata, l'effetto è il seguente:

```
Exception in thread "main" NonToccareLaMiaRobaException
        at Armadietto.apriArmadietto(Armadietto.java:11)
        at Spogliatoio.main(Spogliatoio.java:10)
```

- Su schermo viene stampato un 'riassunto' associato all'eccezione non catturata, chiamato stack trace
- Questo riporta:
  - Il thread in cui l'eccezione è stata sollevata
  - Il nome dell'eccezione sollevata
  - La successione, in ordine inverso di invocazione, dei metodi coinvolti
  - Il file sorgente e il numero di riga di ciascuna invocazione

I metodi printStackTrace() e getMessage()

- L'output generato a schermo da un'eccezione non catturata è prodotto dal metodo **printStackTrace**(), offerto dalla classe Throwable
- Un altro metodo messo a disposizione dalla stessa classe è **getMessage**(), in grado di restituire, se prevista o disponibile, una descrizione sintetica della ragione per la quale si è verificata l'eccezione

```
public class Invenzione
{
    public static void main(String[] args)
    {
        String invenzione = "CappelloPensatore";
        invenzione.charAt(-1);
    }
}
```

```
Exception in thread "main" java.lang.StringIndexOutOfBoundsException: String index out of range: -1
        at java.lang.String.charAt(String.java:694)
        at Invenzione.main(Invenzione.java:8)
```

Creare eccezioni personalizzate

- E’ possibile definire delle eccezioni personalizzate
- In questo modo i messaggi di errore conservano la semantica legata all’applicazione
- Al momento della creazione di un nuovo tipo di eccezione sarà opportuno studiarne la natura e lo scopo (errore sincrono/asincrono, possibilità di recovery o errore irreversibile…) e scegliere la super-classe più adeguata

Come si crea un'eccezione?

- Vediamo la efinizione dell'eccezione personalizzata NonToccareLaMiaRobaException:

```
public class NonToccareLaMiaRobaException extends Exception
{

}
```

Tramite la parola chiave extends è possibile creare una nuova eccezione a partire da un tipo già esistente

La parola chiave throw


```
public class Armadietto
{
    private Sportivo proprietario;
    private boolean aperto;

    public Armadietto(Sportivo proprietario){ this.proprietario = proprietario; }

    public void apriArmadietto(Sportivo g) throws NonToccareLaMiaRobaException, ArmadiettoGiaApertoException
    {
        if (!proprietario.equals(g)) throw new NonToccareLaMiaRobaException();

        if( aperto ) throw new ArmadiettoGiaApertoException();
    }
}
```

- Tramite la parola chiave **throw** è possibile **sollevare** (o lanciare) una nuova eccezione

L'importanza di creare eccezioni personalizzate (1)

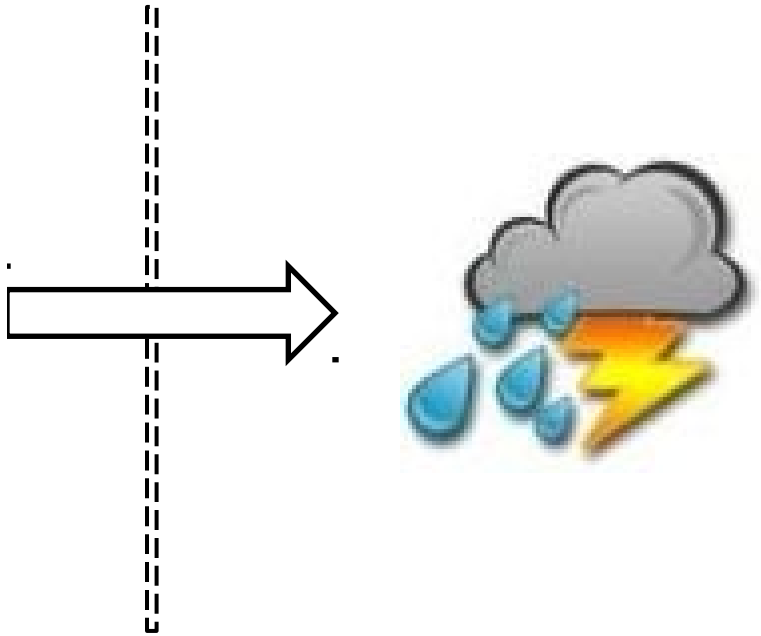
- Consideriamo la classe Scaffale, implementata tramite array:

```
public class Scaffale
{
    private Libro[] libri = new Libro[20];

    public Libro getLibro(int i) throws LibroMancanteException
    {
        if (i<0 || i>=libri.length) throw new LibroMancanteException();

        return libri[i];
    }
}
```

- Immaginiamo che venga invocato il metodo getLibro(50)
- Viene sollevata l'eccezione personalizzata LibroMancanteException,
- In assenza di questa sarebbe stata sollevata l'eccezione IndexOutOfBoundsException…

L'importanza di creare eccezioni personalizzate (2)

- Che significato specifico trasmette IndexOutOfBoundsException ad un utilizzatore di una libreria? **Nessuno**!
- L'eccezione personalizzata, invece, **nasconde** i dettagli implementativi e trasmette un significato appropriato al contesto



Il blocco finally (1)


```
public class EpisodioDisney
{
    public static void main(String[] args) throws EventoMalignoException
    {
        Deposito deposito = new Deposito();
        BandaBassotti bb = new BandaBassotti();
        Amelia amelia = new Amelia();
        Paperino paperino = new Paperino();
        ZioPaperone zioPaperone = new ZioPaperone();
        Battista battista = new Battista();

        try
        {
            // Nel deposito entra Zio Paperone
            deposito.lasciaEntrare(zioPaperone);

            // Entrano dei nemici
            deposito.lasciaEntrare(bb);
            deposito.lasciaEntrare(amelia);

            // Paperino chiedera' di certo soldi...
            zioPaperone.riceviInVisita(paperino);
        }
        catch (AttaccoAlDepositoException e)
        {
            deposito.attivaSistemiDiSicurezza();
        }
        catch (EventoMalignoException e)
        {
            zioPaperone.chiediAiuto(battista);
        }
        finally
        {
            System.out.println("Chiama la polizia!");
        }
    }
}
```

- E' uno speciale blocco posto dopo tutti i blocchi try-catch
- Eseguito **a prescindere** dal sollevamento di eccezioni
- Le istruzioni all'interno del blocco finally vengono **sempre** eseguite (perfino se nel blocco try-catch vi è un return, un break o un continue)
- Unico controesempio: uscita forzata (ad es. tramite System.exit())

blocco finally

Il blocco finally (2)

- Tipicamente all'interno del blocco finally vengono eseguite operazioni di clean-up (es. chiusura di eventuali file aperti o rilascio di risorse) in modo da garantire un certo stato dell'esecuzione

```
public class FileAperto
{
    public static void main(String[] args)
    {
        FileReader fileReader = null;

        try
        {
            fileReader = new FileReader(new File("my/favourite/path"));
            fileReader.read();
        }
        catch (FileNotFoundException e){ e.printStackTrace(); }
        catch (IOException e){ e.printStackTrace(); }
        finally
        {
            try
            {
                fileReader.close();
            }
            catch (IOException e){ e.printStackTrace(); }
        }
    }
}
```

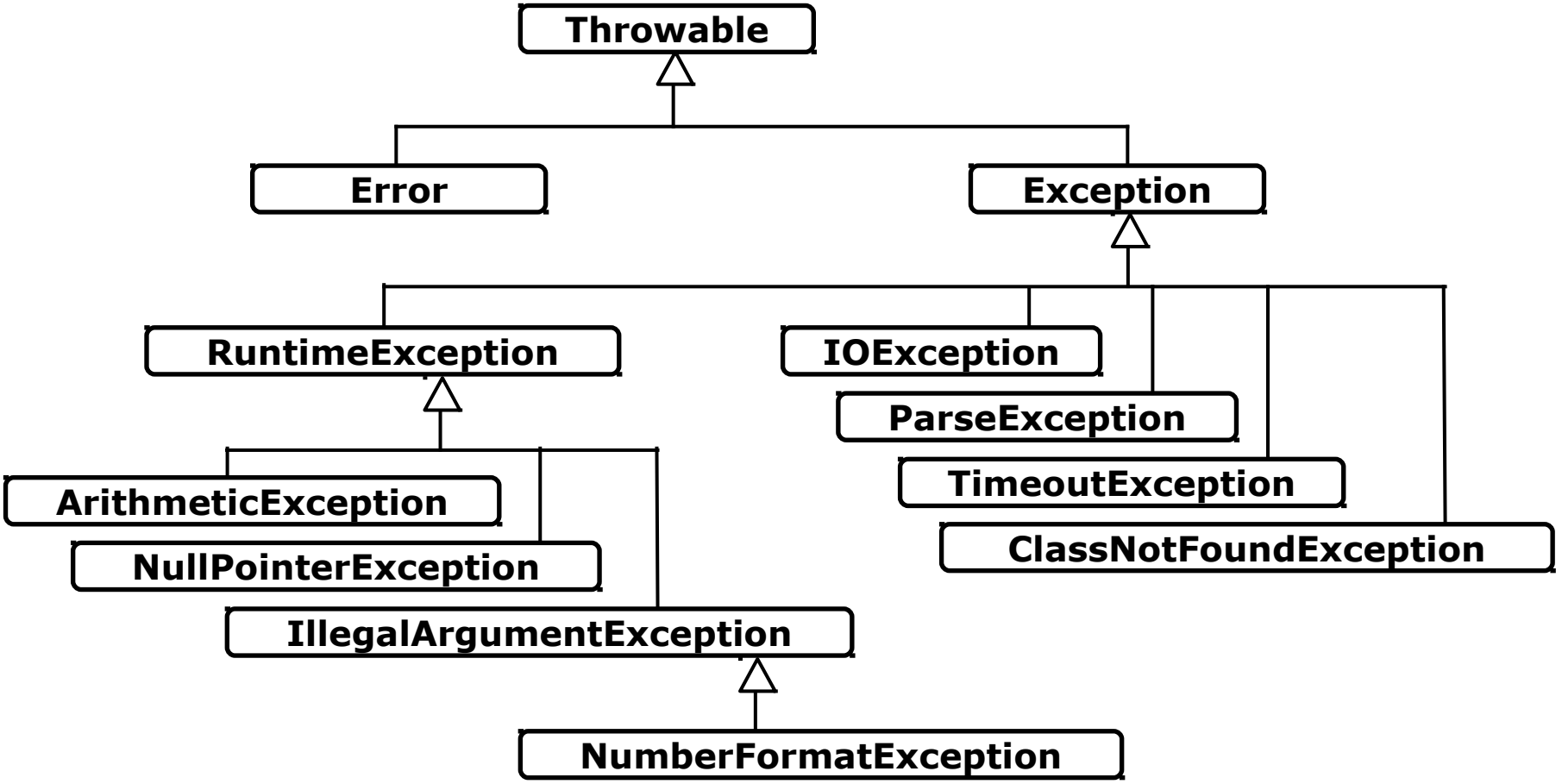
Clean-up!

La classe Throwable

- La classe che implementa il concetto di eccezioni è Throwable che estende direttamente la classe Object
- Gli oggetti di tipo Throwable sono gli unici oggetti che è possibile utilizzare con il meccanismo delle eccezioni

La gerarchia di eccezioni in Java

- Come dirette sottoclassi di Throwable troviamo la classe **Error** e la classe **Exception**:



Le classi Exception e Error

- **Exception**:
  - eccezioni interne alla JVM (classe **RuntimeException**): legate ad errori nella logica del programma
  - eccezioni regolari (es. IOException, ParseException, TimeoutException): errori che le applicazioni dovrebbero anticipare e dalle quali poter riprendersi

- **Error**: cattura l'idea di condizione eccezionale irrecuperabile
  - Assai **rari** e non dovrebbero essere considerati dalle applicazioni (es. ThreadDeath, OutOfMemoryError, …)

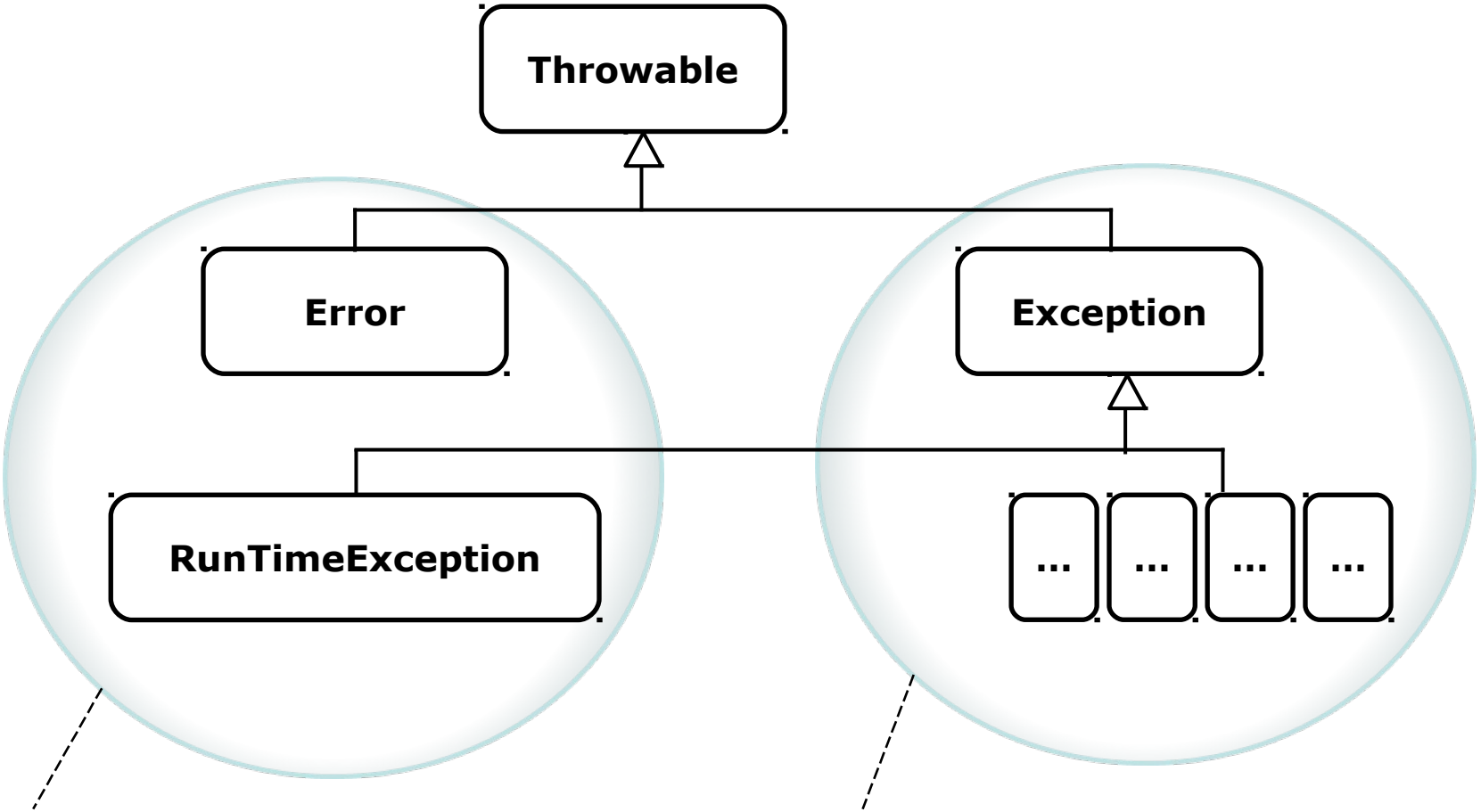
Eccezioni checked e unchecked (1)

- Eccezioni di tipo **checked**:
  - È sempre necessario attenersi al paradigma catch-or-declare
  - Sono eccezioni che estendono Exception (ma non RuntimeException)
  - Esempi: *ParseException, ClassNotFoundException, FileNotFoundExcep*

Eccezioni checked e unchecked (2)

- Eccezioni di tipo **unchecked**:
  - Non si è obbligati a dichiarare le eccezioni sollevate o a catturarle in un blocco try-catch (ma è possibile farlo)
  - Sono eccezioni che estendono Error o RuntimeException
  - Esempi: *IndexOutOfBoundsException, ClassCastException, NullPointerException, ArithmeticException, OutOfMemoryError*

Eccezioni checked e unchecked (3)



Uncheck ed

Checked

(catch-or-declare obbligatorio)

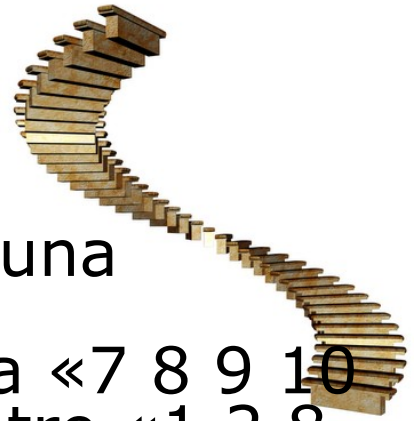
Rilanciare le eccezioni (rethrowing)

- E' possibile inoltre rilanciare eccezioni dall'interno di un blocco catch:

```
public class Ladro
{
    public void apriArmadietto(Armadietto a) throws FurtoException
    {
        throw new FurtoException();
    }
}


public class Spogliatoio
{
    public static void main(String[] args) throws RotturaDiScatoleException
    {
        Ladro ladro = new Ladro();
        Armadietto armadietto = new Armadietto();

        try
        {
            ladro.apriArmadietto(armadietto);
        }
        catch (FurtoException e)
        {
            throw new RotturaDiScatoleException();
        }
    }
}
```

Nuova eccezione sollevata: in questo caso la precedente eccezione viene gestita (e ogni informazione ad essa associata viene persa) e quella corrente diventa la nuova eccezione

Esercizio: Sequenza a gradini



- Definiamo una sequenza "a gradini" come una successione in cui ciascun elemento dista esattamente 1 dal precedente (la sequenza «7 8 9 10 11 12 11 10 11 10 9 8 7» è a gradini, mentre «1 2 8 7 6 5 42 9 20» non lo è)
- Costruire una sequenza a gradini inizialmente vuota
- Inserire successivamente altri elementi nella sequenza tramite il metodo aggiungi(int x)
- Nel caso il prossimo numero aggiunto violi il vincolo della sequenza a gradini, va notificato un **errore** al metodo chiamante
- Prevedere inoltre un metodo che stampi la sequenza finora memorizzata. La sequenza, inoltre, non ha vincoli di lunghezza (potenzialmente infinita)

Esercizio: Floppy disk 2.5’’ (1)

- Implementare un floppy disk da 2.5 pollici
- Il floppy disk è un supporto magnetico che contiene dati e può essere acceduto in lettura e scrittura
- Esso ha una capacità pari a 1.474.560 bytes (ovvero 1.44 MB)
- Il floppy disk possiede anche un blocco scrittura che è possibile attivare o disattivare; questo meccanismo, se attivato, impedisce la scrittura di dati

Esercizio: Floppy disk 2.5'' (2)

- Implementare inoltre i seguenti metodi:
  - posizionaTestina(): posiziona la testina in posizione k-esima
  - leggi(): legge i prossimi x byte
  - scrivi(): scrive i byte passati in input
  - formatta(): cancella tutti i dati presenti sul floppy disk
  - attivaBloccoScrittura(): attiva il blocco scrittura
  - disattivaBloccoScrittura(): disattiva il blocco scrittura
- Gestire inoltre tutte le situazioni di errore, ad esempio:
  - se si cerca di scrivere o formattare mentre è presente il blocco scrittura
  - se si cerca di scrivere ma il disco è pieno
  - si cerca di leggere ma non sono presenti sufficienti dati sul disco

Esercizio: Dizionario e Mappa

- Scrivere un'interfaccia **Dizionario** dotata dei metodi:
  - Elemento search(Chiave k): **cerca** l'elemento associato alla chiave k nella struttura dati
  - void add(Chiave k, Elemento e): **aggiunge** l'elemento e con chiave k nella struttura dati
  - Elemento delete(Chiave k): **rimuove** l'oggetto associato alla chiave k dalla struttura dati
  - int size(): restituisce la **taglia** del dizionario
- Implementare una coppia (k, e) chiave-elemento, costruita a partire da una Chiave k ed un Elemento e ad essa associato
- Implementare la struttura dati Mappa che rappresenta una collezione di coppie senza ripetizione di chiavi
- Prevedere il sollevamento delle seguenti eccezioni:
  - ElementNotFoundException: lanciata nel caso la chiave da cercare o rimuovere non è contenuta nella struttura dati
  - ElementAlreadyContainedException: lanciata nel caso in cui la chiave da aggiungere all'insieme sia già contenuta

Esercizio: Catena di volontari (1)

- Nel 1825 a causa del Miramichi Fire in Canada morirono 160 persone
- La squadra di volontari FIRE (Fuoco Immaginario o Reale Estinguesi) è specializzata nell'estinguere qualunque tipo di fuoco, sia esso fuoco fatuo o un devastante incendio naturale
- Dal gruppo emergono per competenza due figure:
  - La volontaria Acquafredda: la volontaria più veloce nel fornire secchi d'acqua al resto del gruppo
  - Il volontario Fuoco: il volontario dalla mira migliore e in grado di spegnere anche le fiamme più alte
- Il resto del gruppo è formato da semplici volontari uniti in una catena umana, ciascuno in grado di passare secchi d'acqua dal volontario alla sua sinistra a quello alla sua destra

Esercizio: Catena di volontari (2)

- I volontari hanno a disposizione un **unico** secchio d'acqua che è possibile riempire o svuotare
- La volontaria Acquafredda è **l'unica** ad aver **accesso** all'acqua, ovvero l'unica in grado di riempire il secchio
- Il volontario Fuoco invece è **l'unico** in grado di osservare in maniera diretta l'incendio, l'unico a poterlo spegnere e l'unico anche a determinarne l'avvenuta estinzione
- All'incendio è associato un intero compreso tra 1 e 10 che ne determina l'intensità nella scala "Fireneit" e corrisponde, sperimentalmente, al numero di secchi necessari alla sua estinzione

Esercizio: Catena di volontari (3)

- Quando l'incendio si estingue il volontario Fuoco si accorge dell'evento FuocoEstintoException e, secondo il codice condiviso all'interno della FIRE, lancia il messaggio 'BastaAcquaException!'
- Tale messaggio è molto importante ed è necessario che arrivi alle orecchie della volontaria Acquafredda il prima possibile
- Al ricevimento di tale messaggio, la volontaria Acquafredda dovrà smettere di riempire il secchio d'acqua, risorsa assai preziosa. Un obiettivo della missione dev'essere quello di non sprecare acqua ed utilizzarne la giusta quantità

Esercizio: Catena di volontari (4)

- Costruire una catena di volontari avente in testa la volontaria Acquafredda, in coda il volontario Fuoco e in mezzo 3 volontari comuni
- I volontari avranno a disposizione un unico secchio, inizialmente messo a disposizione dalla volontaria Acquafredda, che viene passato da volontario a volontario tramite il metodo void estinguiIncendio(Secchio s)
- Appiccare un Incendio doloso e mettere in moto la catena FIRE in modo che estingua l'incendio