ECCEZIONI IN JAVA

Gli argomenti trattati sono proposti anche nei video dal numero 43 al numero 50 alla seguente playlist di youtube: https://youtube.com/playlist?list=PL-NrWrNHrd0qHhtrcR7KhBW_MPhNeX6u9&feature=shared

Cosa si intende per eccezione?

In Java un'eccezione è un evento che si verifica a causa di una situazione anomala (un errore) durante l'esecuzione (ossia a ***runtime***).

Un'eccezione può avvenire per diversi motivi, ad esempio un'operazione aritmetica impossibile da svolgere (divisione per zero) oppure il tentativo di accedere ad un elemento di un array fuori dai suoi limiti.

Affinchè un programma sia stabile, il programmatore è tenuto a scrivere il codice "prevedendo" gli errori a runtime che si possono verificare e gestendoli, ossia è tenuto a scrivere nel codice le istruzioni che devono essere eseguite nel caso si verifichino tali errori (detti appunto eccezioni). Per chi si ricorda, questo è il concetto di "completezza" degli algoritmi.

Fino ad ora abbiamo sempre gestito i possibili errori con la struttura di controllo "selezione" (IF), ad esempio per una divisione fra numeri reali:

```
 if (denominatore==0)
         comunica "divisione impossibile"
else
         risultato = numeratore/denominatore;
```

Java mette a disposizione un apposito **meccanismo**, molto potente, per la gestione delle eccezioni.

Ma non si potrebbe continuare a gestire le eccezioni con gli IF…..ELSE? Perché dovremmo utilizzare il meccanismo delle eccezioni? Quale è il vantaggio?

E' possibile continuare a gestire gli errori con la selezione, ma nella OOP è molto più conveniente e opportuno utilizzare il meccanismo delle eccezioni.

Il vantaggio delle eccezioni lo vedremo dopo con un esempio, ma è importante dire che tale vantaggio deriva dal fatto che esso è un meccanismo, messo a disposizione

dal compilatore java, che consente di “separare” la parte di codice in cui si genera l’errore dalla parte di codice in cui lo si gestisce.

Con il meccanismo delle eccezioni, infatti, ogni volta che si scrive un metodo all’interno del quale il programmatore sa che si può verificare una errore a run time, si può decidere di agire in due modi:

1. gestire l’eccezione direttamente nel codice di quel metodo

2. “far gestire l’eccezione” al metodo “chiamante” (quello che ha invocato il metodo in cui si è verificata l’eccezione).
   A sua volta, nel metodo “chiamante”, il programmatore potrà decidere nuovamente se “gestire” l’eccezione oppure “passare la gestione” al metodo che ha invocato il metodo “chiamante” (quindi al “chiamante” del “chiamante”)….e così via fino ad arrivare al metodo di partenza (da cui hanno origine le chiamate a tutti i metodi) ossia al metodo main.

Se nessun metodo gestisce l’eccezione, essa arriva dunque al main, e se neppure nel main viene gestita, l’esecuzione del software si interrompe in maniera anomala e viene mostrato un messaggio, chiamato **stack trace**, che indica in successione tutti i metodi (che fanno parte dello stack) attraverso i quali **“è passata”** l’eccezione senza essere gestita. Il programmatore deve di evitare che un’eccezione “si propaghi” fra i metodi senza essere mai gestita fino a causare l’interruzione anomala dell’esecuzione.

Esempio di eccezione non gestita (p. A93- A94 del libro):

ESEMPIO

Il metodo *main* della seguente classe Java ha il solo scopo di generare un’eccezione di divisione per zero:

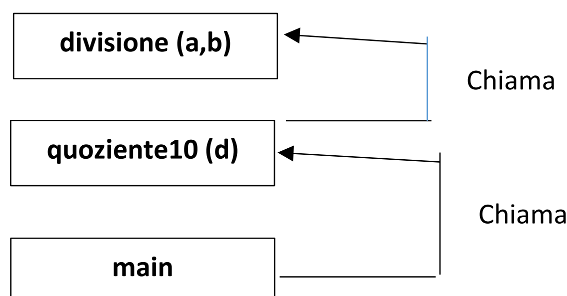
```
1  public class Eccezione {
2
3    static int divisione(int a, int b) {
4      return a/b;
5    }
6
7    static int quoziente10(int d){
8      return divisione(10, d);
9    }
10
```

```
11    public static void main (String[] args) {
12      for (int n=10; n>=0; n--)
13        System.out.println(quoziente10(n));
14    }
15  }
```

Il messaggio di output prodotto al verificarsi dell'eccezione (quando la variabile *n* assume valore 0) specifica la catena di invocazione del metodo che ha causato l'eccezione stessa; in questo caso il metodo *divisione* invocato dal metodo *quoziente10*, a sua volta invocato dal metodo *main*:

```
1
1
1
1
1
2
2
3
5
10
Exception in thread "main" java.lang.ArithmeticException: / by zero
   at Eccezione.divisione(Eccezione.java:4)
   at Eccezione.quoziente10(Eccezione.java:8)
   at Eccezione.main(Eccezione.java:13)
```

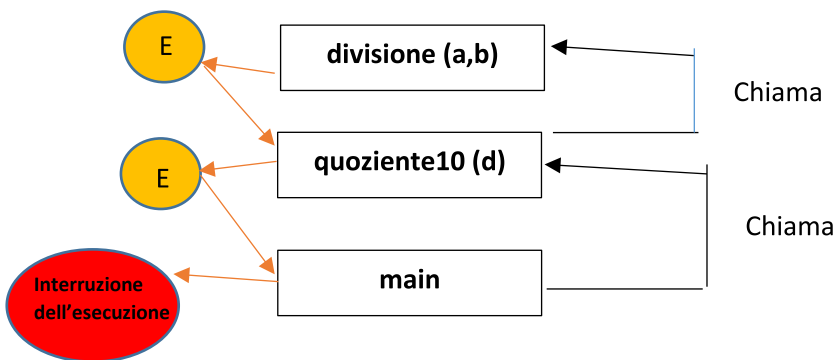
Nell'esempio, lo stack di chiamate dei metodi è il seguente:



Si osservi che il metodo main invoca per 11 volte il metodo "quoziente10" (che a sua colta invoca il metodo "divisione") ma solo all' 11esima volta si verifica l'eccezione (quando il divisore è =0).

Quindi all'11esima chiamata del metodo "divisione" si genera, in tale metodo, un'eccezione (in questo caso è un'eccezione del tipo "ArithmeticException"…poi vedremo le altre….), e il percorso dell'eccezione è il seguente:

- L’eccezione viene generata (anzi, il termine corretto è “**viene sollevata**”) in “divisione” perché al metodo viene chiesto di svolgere una operazione impossibile (10/0)
- L’eccezione non viene gestita in “divisione” per questo viene “passata” a “quoziente10”
- L’eccezione non viene gestita in “quoziente10” per questo viene “passata” al “main”
- L’eccezione non viene gestita nel “main” quindi l’esecuzione del programma viene interrotta “in maniera anomala”.



L’output mostra lo stack trace che riporta, nell’ordine, tutti i metodi “attraversati” dall’eccezione

Ma perché JAVA consente che un’eccezione venga “passata” ed eventualmente gestita da uno dei metodi chiamanti, anziché dal metodo in cui si è verificata? C’è un’utilità in questo?

Certo! Come sappiamo, il vantaggio della OOP è il riutilizzo del codice, quindi un qualsiasi metodo di una classe può essere riutilizzato in molti diversi software.

Quando si verifica un’eccezione in un metodo, tale metodo potrebbe non avere “tutte le informazioni” su come gestire l’eccezione, invece il metodo “chiamante” (o il “chiamante del chiamante”) “sapendo” in quale contesto è avvenuta la chiamata, può gestire al meglio l’eccezione.

Chiariamo il concetto richiamando l’esempio precedente ed inserendolo nel contesto di una applicazione.

Supponiamo che il metodo “divisione” venga utilizzato in un software “non molto importante”, ad esempio un software che serve per calcolare la media dei tempi di

percorrenza di un programma di allenamento e stamparla su un file. In questo caso un’eventuale eccezione dovuta a una divisione per zero potrebbe **essere gestita nel main**, quindi nel metodo chiamante, semplicemente visualizzando un messaggio di errore da stampare su file (“calcolo non possibile”), dopodiché l’esecuzione potrebbe proseguire. Quindi l’applicazione, in questo caso, non verrebbe interrotta in modo anomalo e, in caso di errore, nel file si ritroverebbe il testo “calcolo non possibile”.

Supponiamo ora un’altra situazione, in cui il metodo “divisione” viene utilizzato in un software che si occupa di dosare i medicinali nella farmacia di un ospedale, in tal caso un’eventuale divisione per zero potrebbe portare a dosi errate di medicinale. In questo secondo caso è opportuno che nel main tale situazione venga gestita in un altro modo (verranno attivati allarmi, verrà fermata la preparazione del medicinale ecc..).

Quando un programmatore scrive il metodo “divisione” non può sapere in quale software esso sarà poi riutilizzato, per questo è opportuno che l’eccezione venga gestita dal metodo chiamante, dove il programmatore saprà “cosa far fare al programma” quando l’eccezione si verifica (chiudere dei file, chiudere delle connessioni di rete, comunicare dei messaggi..).

Un secondo motivo per gestire l’eccezione in un punto diverso rispetto a dove essa è stata sollevata è che ciò rende generalmente più leggibile e ordinato il codice.

Ma tutto questo non potrei farlo anche con gli IF?

Certo, ma con gli IF il programmatore deve scrivere ogni volta la struttura (il codice) che esegue il passaggio al metodo chiamante di un eventuale caso che può generare una eccezione. Nel nostro esempio della funzione divisione dunque il codice dovrebbe essere:

```
IF (divisore ==0)
        return -1    // ‘poi nel metodo “chiamante” bisognerà gestire questo -1 con un altro IF
                     per passarlo al “chiamante” del “chiamante” (IF //(divisione)==-1…..) e così
                     via…  (E poi c’è un problema: e se il risultato della divisione fosse realmente
                     -1?...)
else
        risultato = num/den;
```

Il programmatore deve costruirsi un “proprio codice” dove -1 significa qualcosa, -2 qualcos’altro ecc.

Le eccezioni invece costituiscono un meccanismo comodo, già pronto, per il passaggio di eventuali errori a *runtime* fra i metodi chiamanti.

Ma quando si dice che l’eccezione non viene gestita da un metodo ma viene “passata” al metodo chiamante, cosa è esattamente che viene passato?

In Java le eccezioni sono **oggetti**. Essi sono istanze di apposite classi predefinite. Le eccezioni sono di diverso tipo (vedremo le principali…). Grazie al meccanismo dell’ereditarietà tutte le eccezioni derivano (ossia sono figlie) dalla classe ***Exception*** che deriva dalla classe ***Throwable*** che è compresa nel package ***java.lang***, già importato di default nei principali IDE Eclipse e netBeans.

Quando si dice che viene “passata” un’eccezione si intende quindi che un metodo restituisce un oggetto “eccezione” al metodo chiamante, proprio come se facesse un “return”. Possiamo quindi affermare che il meccanismo delle eccezioni fornisce un modo alternativo (al return) per terminare l’esecuzione di un metodo.

Un metodo dunque può terminare la propria esecuzione in due modi:

1. “arrivando” all’ultima istruzione prevista restituendo al chiamante un valore (con return…), che può anche essere void.

2. Interrompendosi ad un certo punto dell’esecuzione restituendo un’eccezione.

Come si gestisce un’eccezione.

Come abbiamo detto, quando si verifica un’eccezione all’interno di un metodo, il programmatore può scegliere se gestire l’eccezione all’interno di quel metodo oppure “passarla” al metodo chiamante e gestirla in quest’ultimo metodo. Prima o poi comunque, in qualche metodo, se si vuole realizzare un’applicazione stabile, l’eccezione andrà gestita, altrimenti l’applicazione, al verificarsi dell’eccezione, si interromperà in maniera anomala, e questo va assolutamente evitato.

Ora vedremo qual è il costrutto (la sintassi) che consente la gestione dell’eccezione, poi vedremo invece come si fa a “passare” l’eccezione al chiamante.

Una volta individuato il punto del codice in cui si decide di gestire l’eccezione, essa andrà gestita con il costrutto **try..catch…,** che costituisce una specie di “trappola” per gli errori.

```
try {
  // istruzioni che potenzialmente generano eccezioni
  …;
  …;
  …;
}
catch (TipoEccezione1 identificatore){
  // istruzioni da eseguire se si verifica l'eccezione
  // TipoEccezione1
  …;
  …;
  …;
}
catch (TipoEccezione2 identificatore){
  // istruzioni da eseguire se si verifica l'eccezione
  // TipoEccezione2
  …;
  …;
  …;
}
finally {
  // istruzioni da eseguire per un qualsiasi tipo di eccezione
  …;
  …;
  …;
}
```

E' la classe a cui appartiene l'eccezione

E' il nome dato a questa eccezione, lo sceglie il programmatore

All'interno del blocco **try** si scrivono le istruzioni che potenzialmente possono generare delle eccezioni (poche istruzioni, una, due o tre generalmente).

Nel caso si verifichi un'eccezione, l'esecuzione passa al blocco **catch** corrispondente. All'interno del blocco si scriveranno dunque le istruzioni per la gestione dell'eccezione, ossia le istruzioni da eseguire nel caso si verifichino le eccezioni (ad esempio: comunica l'errore, chiudi un file, non fare nulla, ecc..).

Il blocco **finally** è opzionale (non obbligatorio), contiene delle istruzioni che verranno **sempre eseguite** sia nel caso si dovesse verificare una qualsiasi eccezione (sia una eccezione prevista in un catch, sia non prevista), sia nel caso non si dovesse verificare alcuna eccezione.

Poiché può darsi che in un blocco try si possano verificare più tipi di eccezioni diverse (divisione per zero, puntatore a null…..), quindi va specificato un blocco catch per ciascun tipo di eccezione che si può verificare. Si noti anche che per ciascun tipo di eccezione gestita in un blocco catch, viene assegnato un identificatore (un identificatore non è altro che un nome, ad esempio: eccezioneDivisionePerZero, eccezionePuntatoreNull, o semplicemente "exception" o "e")

Vediamo come può essere gestita l'eccezione ArithmeticException dell'esempio precedente.In questo caso il programmatore decide di gestire l'eccezione nello stesso metodo in cui essa viene sollevata (nel metodo "divisione"). La gestione consiste in un semplice messaggio di output "divisione a/b impossibile".

Si osservi che per gestire l'eccezione, in questo caso, è necessario modificare da int a String il tipo restituito dai due metodi "divisione" e "quoziente10".

```
1  public class Eccezione {
2
3    static String divisione(int a, int b) {
4      try{
5        return Integer.toString(a/b);
6      }
7      catch(ArithmeticException exception) {
8        return "impossibile calcolare "+a+"/"+b;
9        }
10     }
11
12   static String quoziente10(int d){
13     return divisione(10, d);
14   }
15
16   public static void main (String[] args) {
17     for (int n=10; n>=0; n--)
18       System.out.println(quoziente10(n));
19   }
20 }
```

L'output che si ottiene è il seguente e il programma non termina in questo caso l'esecuzione in modo anomalo:

```
1
1
1
1
1
2
2
3
5
10
impossibile calcolare 10/0
```

Infatti, al verificarsi dell'eccezione di classe *ArithmeticException* (*exception* è un oggetto istanza di questa classe specifico per l'errore rilevato), questa viene intercettata e il codice del blocco `catch` costruisce in questo caso il messaggio di risposta appropriato.

Come si "passa al chiamante" un'eccezione.

Bisogna ora fare una classificazione delle eccezioni perché a seconda del tipo di eccezioni il programmatore può comportarsi in maniera diversa.

Le eccezioni si suddividono in due grandi categorie, le eccezioni **unchecked** e le eccezioni **checked**.

Eccezioni di tipo unchecked (non controllate):

Un esempio di eccezione predefinita uncheked è quella appena vista (ArithmeticException). Questo tipo di eccezioni sono così chiamate perché JAVA **non ne rende obbligatoria gestione nel codice**, ma lascia al programmatore la facoltà di gestirle o meno. Anche se l'eccezione non è gestita nel codice, il compilatore consentirà comunque la compilazione. Naturalmente se poi, a run time, l'eccezione si verifica e la gestione non è presente, l'esecuzione si interrompe con un errore.

Quindi se l'eccezione è uncheked non è necessaria alcuna istruzione per "passare" l'eccezione al chiamante, semplicemente quando essa si verifica, o viene gestita nel metodo in cui si è generata, oppure passa "automaticamente" al metodo chiamante (e poi al chiamante del chiamante…) fino a quando uno dei metodi chiamatnti si occupa di gestirla con un try catch. Se nessun metodo (neppure il main) gestisce l'eccezione con un try catch, l'esecuzione verrà interrotta in maniera anomala e verrà mostrato lo stack trace (cosa da evitare assolutamente).

Nell’esempio precedente, essendo l’eccezione di tipo unchecked, il programmatore avrebbe potuto gestire l’eccezione con un try catch in uno dei due metodi di livello superiore (i chiamanti: “quoziente10” o “main”) rispetto al metodo in cui essa si può verificare (“divisione”).

Soluzione 1: l’eccezione viene gestita nel metodo chiamante (quoziente10)

Gestione dell’eccezione nel metodo “chiamante” (quoziente10)

```java
public class Eccezione
{
    public static int divisione(int a, int b)
    {
            return a/b;
    }
    public static String quoziente10 (int x)
    {
        try
        {
            return Integer.toString(divisione(10,x));
        }
        catch (ArithmeticException eccezione)
        {
            return ("Quoziente di 10 diviso per "+x+" impossibile da calcolare");
        }
    }
    public static void main(String[] args)
    {
        for (int n = 10; n >=0; n--)
        {
            System.out.println(quoziente10(n));
        }
    }
}
```

Soluzione 2: gestione nel metodo chiamante del chiamante (main)

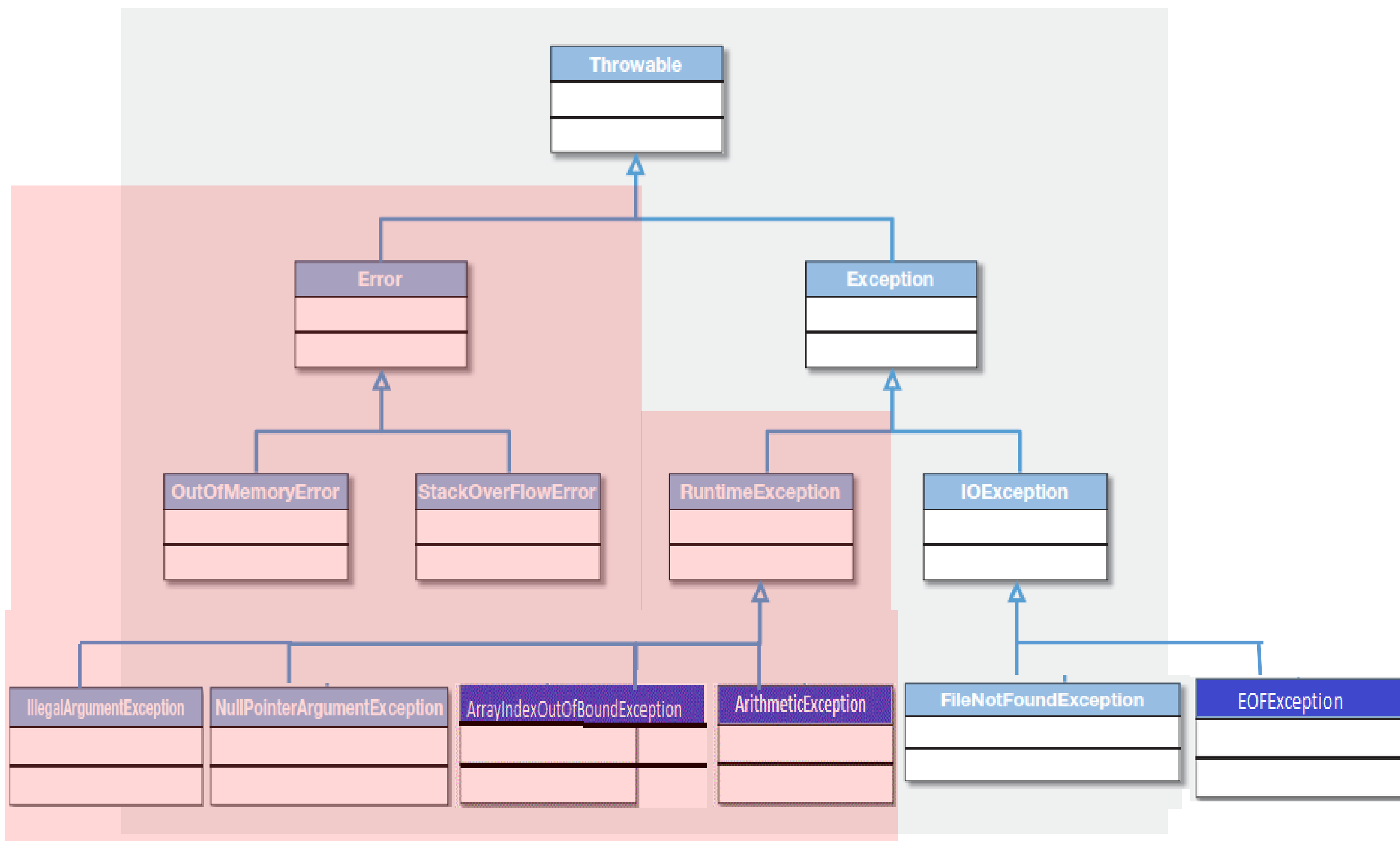
```
public class Eccezione
{
    public static int divisione(int a, int b)
    {
            return a/b;
    }
    public static int quoziente10 (int x)
    {
        return divisione(10,x);
    }
    public static void main(String[] args)
    {
        for (int n = 10; n >=0; n--)
        {
            try
            {
                System.out.println(quoziente10(n));
            }
            catch (ArithmeticException eccezione)
            {
                System.out.println("Per n="+n+" l'operazione è impossibile.");
            }
        }
    }
}
```

Gestione dell'eccezione nel metodo "chiamante" del "chiamante" (main)

La scelta di quale metodo debba gestire l'eccezione dipende dal singolo caso, non c'è una regola generale. Ci sono due cose importanti da sottolineare:

1. L'eccezione va prima o poi gestita per evitare che il programma si interrompa in modo imprevisto.
2. Nelle eccezioni di tipo uncheked (come ArithmeticException) non c'è alcun meccanismo che obblighi il programmatore a gestire le eccezioni. La loro gestione o meno è lasciata unicamente all'attenzione del programmatore (vedremo che per le eccezioni checked non è così)
   Perché c'è questa libertà di non gestire le eccezioni? Perché il compilatore sa che un'eccezione di questo tipo (uncheked) potrebbe essere gestita in modo diverso, ad esempio nel nostro caso l'eccezione potrebbe essere evitata controllando che l'input di "quoziente10(int x)" sia sempre diverso da 0.

Vediamo quali sono le eccezioni unchecked predefinite di java (quelle in “rosso”):

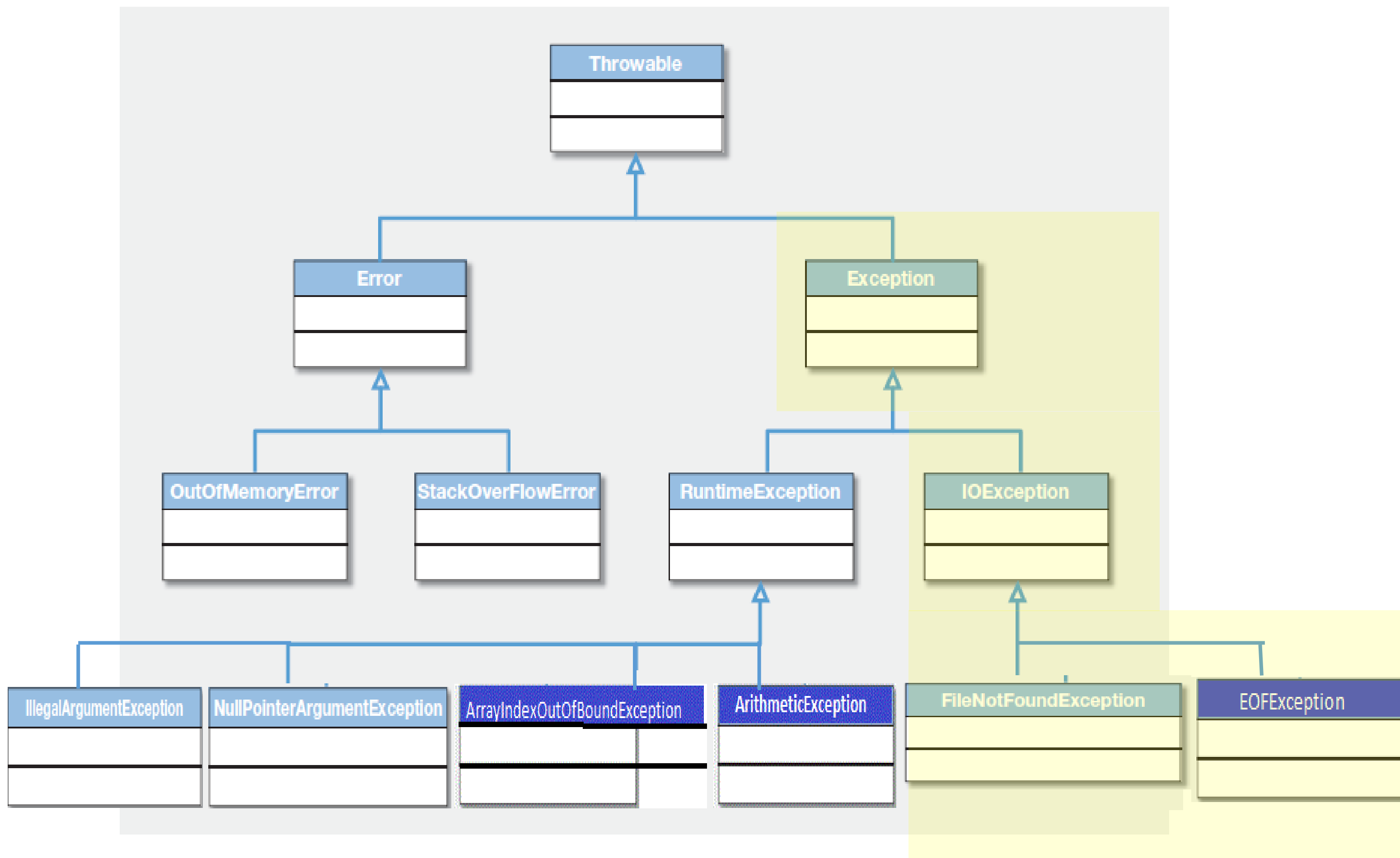


- Gli Errori (oggetti figli della classe **Error**):
  sono generati da eventi esterni all’applicazione, indipendenti dal programmatore (rottura dell’ HD, interruzione di una connessione di rete….), l’unica gestione possibile è quella di comunicare l’errore con un messaggio. La responsabilità del verificarsi di un errore non è del programmatore.

- Le eccezioni figlie della classe **RuntimeException**:
  Sono errori dovuti alla programmazione non corretta, la responsabilità è del programmatore (**bisogna saperli**)
  - **IllegalArgumentException** (passaggio di parametri ad un metodo non accettati) (generalmente viene sollevata “volontariamente” dal programmatore quando i parametri passati ad un metodo non sono accettabili, per effettuare il controllo dell’input..vedremo)
  - **NullPointerArgumentException** (accesso a un reference che punta a null)
  - **ArrayIndeOutOfBoundException** (“sforamento” array)
  - **ArithmeticException** (errori aritmetici)

Eccezioni di tipo checked (controllate):

Vediamo quali sono le eccezioni checked predefinite di java (quelle in “giallo”):



Queste eccezioni sono chiamate **checked** perché devono essere obbligatoriamente gestite dal programmatore. Come fa Java ad obbligare il programmatore? Semplicemente non compila! Infatti quando il compilatore rileva la presenza di istruzioni che potrebbero generare una eccezione checked, segnala un errore e non compila finché non viene scritto il codice di gestione (try- catch) oppure non viene SPECIFICATAMENTE INDICATA NEL CODICE L’ISTRUZIONE PER IL PASSAGGIO DELL’ECCEZIONE AL METODO CHIAMANTE (vedremo qui di seguito quale è la sintassi per farlo).

Perché queste situazioni devono essere obbligatoriamente gestite? Perché non sono errori, a dire il vero. Sono “situazioni particolari” che si presentano durante l’esecuzione del software (termine della lettura di un file, file non trovato..), non sono errori irrecuperabili (come ad esempio un error dovuto alla rottura dell’ Hard Disk, che invece è unchecked). Java quindi richiede che quando tali eccezioni checked si

verificano, la situazione “anomala” venga “recuperata” con una corretta scrittura del codice.

JAVA obbliga il programmatore a gestire o passare al chiamante l’eccezione, l’importante è che qualche metodo, in ultima istanza il main, dovrà prima o poi gestirla con un try catch.

Alcune eccezioni checked predefinite sono le seguenti. Per ora è opportuno ricordarsi queste, poi se ne incontreranno altre, man mano che verranno introdotti nuovi elementi di programmazione.

- eccezioni predefinite di tipo ioException:
  - **EOFException**
  - **FileNotFoundException**

Ciò che deve essere chiaro è la differenza fra checked (Java obbliga il programmatore a gestirle) e unchecked (possono essere gestite ma Java non obbliga a farlo)

Vediamo un esempio che mostri la gestione di una eccezione di tipo **Cheked**:

Supponiamo, nell’esempio precedente, di voler anche scrivere dei valori in un file di testo, creiamo pertanto un oggetto di classe “WriteFile” ossia un file aperto in scrittura (non preoccupiamoci per ora di come si usa questa classe..vedremo più avanti)

```
import java.io.*;

public class Eccezione
{
    public static int divisione(int a, int b)
    {
        FileWriter fileBackup;      //reference di un oggetto di classe FileWriter (file aperto in scrittura)
        fileBackup=new FileWriter("Z:\\FileCheNonEsiste.txt");   // Istanzio l'oggetto di calsse File

        return a/b;
    }
    public static int quoziente10 (int x)
    {
        return divisione(10,x);
    }
    public static void main(String[] args)
    {
        for (int n = 10; n >=0; n--)
        {
            try
            {
                System.out.println(quoziente10(n));
            }
            catch (ArithmeticException eccezione)
            {
                System.out.println("Per n="+n+" l'operazione è impossibile.");
            }
        }
    }
}
```

Istruzione che genera eccezione FileNotFounException (oppure, più in generale IOException)

Poiché l'istruzione

```
fileBackup=new FileWriter("Z:\\FileCheNonEsiste.txt");
```

potrebbe generare una eccezione di tipo FileNotFoundException, e tale eccezione è checked, Java non permette la compilazione.

Per risolvere il problema i sono due strade (suggerite già dal suggeritore dell'IDE):

1. "Circondare" l'istruzione con un blocco try – catch, e quindi gestire l'eccezione.
2. Indicare specificatamente che l'eccezione verrà gestita dal chiamante. Per fare questo è necessario che nella signature del metodo venga richiamata la keyword **throws** seguita dalla classe dell'eccezione:

La soluzione 1 è uguale al caso delle eccezioni unchecked, la soluzione 2 è la seguente:

Dichiarazione che "passa" l'eccezione al chiamante

```
import java.io.*;

public class Eccezione
{
    public static int divisione(int a, int b) throws IOException
    {
        FileWriter fileBackup;      //reference di un oggetto di classe FileWriter (file aperto in scrittura)
        fileBackup=new FileWriter("Z:\\FileCheNonEsiste.txt");   // Istanzio l'oggetto di calsse File

        return a/b;
    }
    public static int quoziente10 (int x)
    {
        return divisione(10,x);
    }
    public static void main(String[] args)
    {
        for (int n = 10; n >=0; n--)
        {
            try
            {
                System.out.println(quoziente10(n));
            }
            catch (ArithmeticException eccezione)
            {
                System.out.println("Per n="+n+" l'operazione è impossibile.");
            }
        }
    }
}
```

Ora però il problema di gestire l'eccezione ce l'ha il metodo chiamante

Il problema di gestire l'eccezione ora ce l'ha il chiamante (il metodo quoziente10), infatti Java sa che l'invocazione di "divisione(10,x)" può generare una eccezione IOException. Java impedisce la compilazione finchè il programmatore non modifica il codice in uno dei due modi precedentemente visti

1. "circondare" l'istruzione in un blocco try catch
2. "passare" l'eccezione al chiamante (main) con un **"throws"** nella dichiarazione

Attuiamo nuovamente la soluzione 2, a questo punto il problema della gestione dell’eccezione passa al main:

```
import java.io.*;

public class Eccezione
{
    public static int divisione(int a, int b) throws IOException
    {
        FileWriter fileBackup;      //reference di un oggetto di classe FileWriter (file aperto in scrittura)
        fileBackup=new FileWriter("Z:\\FileCheNonEsiste.txt");   // Istanzio l'oggetto di calsse File

        return a/b;
    }
    public static int quoziente10 (int x) throws IOException
    {
        return divisione(10,x);
    }
    public static void main(String[] args)
    {
        for (int n = 10; n >=0; n--)
        {
            try
            {
                System.out.println(quoziente10(n));
            }
            catch (ArithmeticException eccezione)
            {
                System.out.println("Per n="+n+" l'operazione è impossibile.");
            }
        }
    }
}
```

Dichiarazione che “passa” l’eccezione al chiamante

Ora però il problema di gestire l’eccezione IOException ce l’ha il metodo chiamante (main)

Anche qui è possibile applicare la soluzione 1 o la soluzione 2.

Se si applica ancora la soluzione 2 (ossia si passa l’eccezione ancora una volta), l’eccezione risulterà non gestita, e nel caso essa si dovesse verificare, l’esecuzione verrà interrotta mostrando lo stack trace (ma Java ha fatto di tutto per impedirlo!!! A questo punto è proprio il programmatore che NON VUOLE gestirla!). In un buon programma la mancata gestione di un’eccezione va assolutamente evitata.

Soluzione 2:

```
import java.io.*;

public class Eccezione
{
    public static int divisione(int a, int b) throws IOException
    {
        FileWriter fileBackup;      //reference di un oggetto di classe FileWriter (file aperto in scrittura)
        fileBackup=new FileWriter("Z:\\FileCheNonEsiste.txt");   // Istanzio l'oggetto di calsse File

        return a/b;
    }
    public static int quoziente10 (int x) throws IOException
    {
        return divisione(10,x);
    }
    public static void main(String[] args) throws IOException
    {
        for (int n = 10; n >=0; n--)
        {
            try
            {
                System.out.println(quoziente10(n));
            }
            catch (ArithmeticException eccezione)
            {
                System.out.println("Per n="+n+" l'operazione è impossibile.");
            }
        }
    }
}
```

Anche il main "passa" l'eccezione senza gestirla. A questo punto l'eccezione è definitivamente NON GESTITA, Il compilatore permette la compilazione. Nel caso si dovesse verificare l'eccezione, l'esecuzione verrà interrotta e apparirà lo stack trace

Con la soluzione 1, invece, aggiungiamo un catch alla struttura try – catch già presente per aggiungere la gestione dell'eccezione IOException oltre alla ArithmeticEcxeption.

(Oppure potremmo aggiungere una nuova struttura try catch nidificata, oppure una or "IOException" (con una sola |) al catch già presente….tutte le possibili soluzioni vengono "suggerite" dall'IDE, sceglieremo quella che, caso per caso riterremo più opportuna)

Soluzione 1:

```
import java.io.*;

public class Eccezione
{
    public static int divisione(int a, int b) throws IOException
    {
        FileWriter fileBackup;      //reference di un oggetto di classe FileWriter (file aperto in scrittura)
        fileBackup=new FileWriter("Z:\\FileCheNonEsiste.txt");   // Istanzio l'oggetto di calsse File

        return a/b;
    }
    public static int quoziente10 (int x) throws IOException
    {
        return divisione(10,x);
    }
    public static void main(String[] args)
    {
        for (int n = 10; n >=0; n--)
        {
            try
            {
                System.out.println(quoziente10(n));
            }
            catch (ArithmeticException eccezione)
            {
                System.out.println("Per n="+n+" l'operazione è impossibile.");
            }
            catch (IOException eccezione2)
            {
                System.out.println("Errore apertura file di backup");
            }
        }
    }
}
```

Gestione dell'eccezione nel main

Tutte le eccezioni predefinite saranno sono figlie della classe Exception (saranno quindi di tipo checked) oppure figlie della classe RuntimeException (e saranno quindi unchecked).

Questo meccanismo di gestione delle eccezioni **cheked** viene chiamato **handle or declare**, che significa gestisci o dichiara. Con questo termine si indica, dunque, l'obbligo, per il programmatore di gestire un'eccezione cheked con un try..catch (**handle**), oppure di "passare" esplicitamente la gestione al metodo chiamnte con throws nell'intestazione (**declare**) .

Esempio p. A97-A98 (fare e capire)

In questo esempio vengono riscritti i metodi della classe Mensola in modo che le istruzioni che possono generare eccezioni siano inserite in un blocco try-catch. I metodi interessati sono i seguenti:

- setVolume: anziché verificare se la "posizione" è corretta mettiamo le istruzioni di inserimento in un try-catch dove intercetteremo l'eccezione `ArrayIndexOutOfBoundsException`. Nel caso si verifichi l'eccezione, il metodo restituirà -1. (lasciamo gli if per la gestione del caso con posizione occupata).
- getVolume: Anziché verificare la correttezza della posizione e verificare che la posizione non sia "vuota", mettiamo l'istruzione che restituisce il libro in un try-catch ed intercettiamo le eventuali eccezioni (`ArrayIndexOutOfBoundsException`, `NullPointerException`), restituendo null nel caso si verifichi una delle due.
- rimuoviVolume: anziché verificare se la posizione è corretta, si mette in un try catch le istruzioni del blocco, si intercetta l'eventuale eccezione `ArrayIndexOutOfBoundsException`, e in tal caso si restituisce -1.

Oltre alle eccezioni predefinite da Java, il programmatore può "creare" delle proprie classi "eccezione" specificamente create per il proprio progetto, definendo nuove classi che derivano dalla classe Exception, quindi checked.

Definizione e gestione di eccezioni da parte del programmatore

Oltre alle eccezioni predefinite da Java, un programmatore può creare delle **proprie eccezioni**. Per creare una "propria" eccezione il programmatore deve creare una nuova classe e definire tale classe come figlia della classe Exception. La nuova eccezione creata sarà un'eccezioni checked. Per specificare che una classe è figlia della classe Exception, si aggiunge, nell'intestazione della classe, al nome della classe l'indicazione: "**extends Exception**".

L'utilità, nel "creare" nuove eccezioni, sta nel fatto che il programmatore può decidere in quale caso debba sollevarsi l'eccezione. L'eccezione verrà sollevata quando un metodo "non si conclude in maniera corretta", quindi per qualche motivo (ad esempio dati di input non ritenuti validi), il metodo dovrà terminare la propria esecuzione sollevando un'eccezione anziché con un return.

Quando un programmatore crea una propria eccezione, egli dovrà anche scrivere il codice che, al verificarsi di una certa situazione, "solleva" l'eccezione, ossia "istanzia un oggetto della classe eccezione da lui creata".

Per sollevare un'eccezione si utilizza la parola chiave **throw new "nome della classe eccezione"** (occhio a non confondere **throw** con **throws**, quest'ultima va posta nella dichiarazione di un metodo per indicare che un'eccezione viene passata al chiamante).

Esempio:

Si crei una classe Studente, con i seguenti attributi:

- cognome
- nome
- codiceFiscale

La classe possiede i metodi costruttore, getter, setter, toString.

Se il parametro codiceFiscale del costruttore (e del setCodiceFiscale) non è esattamente di 16 cifre, la classe solleva un'eccezione "EccezioneCodiceFiscaleNonValido" che impedisce di istanziare l'oggetto

Codice classe EccezioneCodiceFiscaleNonValido

```
public class EccezioneCodiceFiscaleNonValido extends Exception
{

}
```

La classe è filgia della classe Exception

Codice classe Studente

```
public class Studente
{
    private String Cognome;
    private String Nome;
    private String CF;

    public Studente(String Cognome, String Nome, String CF) throws EccezioneCodiceFiscaleNonValido
    {
        this.Cognome = Cognome;
        this.Nome = Nome;
        setCF(CF);
    }

    public String getCF()
    {
        return CF;
    }

    public void setCF(String CF) throws EccezioneCodiceFiscaleNonValido
    {
        if (CF.length()!=16)
            throw new EccezioneCodiceFiscaleNonValido();
        this.CF = CF;
    }

    public String getCognome() {
        return Cognome;
    }

    public void setCognome(String Cognome) {
        this.Cognome = Cognome;
    }

    public String getNome() {
        return Nome;
    }

    public void setNome(String Nome) {
        this.Nome = Nome;
    }
}
```

3

L'eccezione viene "Passata" al chiamante, che in questo caso è il main della classe APP

1

Se il CF non è valido, viene sollevata l'eccezione

"EccezioneCodiceFiscaleNonValido"

2

L'eccezione viene "Passata" al chiamante, che in questo caso è il costuttore

Codice Classe App:

```
5    package com.mycompany._2_studemteeccezionecf;
6
7    import java.util.logging.Level;
8    import java.util.logging.Logger;
9
10   /**
11    *
12    * @author gian
13    */
14   public class App
15   {
16
17       public static void main(String[] args)
18       {
19           try
20           {
21               Studente s1=new Studente(Cognome: "Pinna",Nome: "Luciano",CF: "357638t7");
22           }
23           catch (EccezioneCodiceFiscaleNonValido ex)
24           {
25               System.out.println(x: "Impossibile istanziare lo studente, il CF non è corretto");
26           }
27       }
28   }
29
```

L'eccezione viene gestita con un try-catch. Se il codice fiscale non è valido l'eccezione viene catturata e gestita. La gestione fa sì che l'oggetto non vemnga istanziato e mostra semplicemente un messaggio all'utente

Esempio refactoring esercizio libreria (fare e capire)

Per capire come funziona la creazione di nostre eccezioni personalizzate, decidiamo di fare un *refactoring* della classe Scaffale. In questo caso il refactoring consiste nel **creare** quattro eccezioni relative ai possibili casi di "situazioni anomale" che si possono verificare a runtime nella classe Scaffale. Le situazioni anomale sono le seguenti

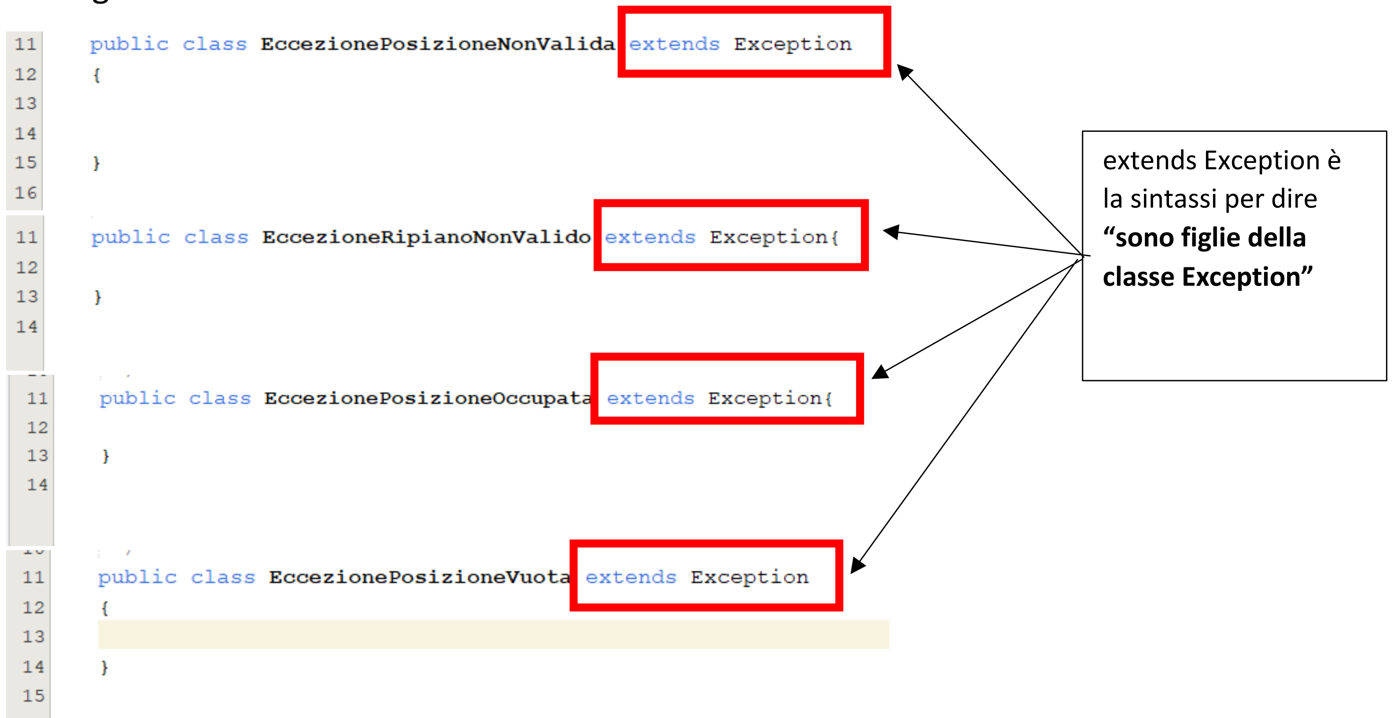
1. accesso ad una posizione non valida (posizione<0 oppure posizione >NUM_MAX_VOLUMI della classe mensola)
2. accesso a un ripiano non valido (ripiano<0 oppure ripiano >NUM_RIPIANI)
3. tentativo di inserimento di un libro in una posizione già occupata.
4. tentativo di ottenere un libro da una posizione vuota

Andiamo a gestire questi 4 possibili "errori a runtime" nelle classi Mensola, Scaffale e App con il meccanismo delle eccezioni "create da noi".

Refactoring passo 1:

Innanzitutto creiamo un **nuovo package e chiamiamolo "eccezioni"**. In questo package andremo a creare le classi "eccezioni", una per ciascuno dei 4 casi.

Tali classi saranno le più semplici possibili, **ossia senza body, solamente con la signature**. E' importante che le classi siano figlie della classe Exception. Il loro codice è il seguente:



Il codice relativo alle modifiche verrà riportato di seguito. Si consiglia di apportare tali modifiche al proprio software "libreria" con l'aiuto del codice sotto riportato e poi riprovare a fare la stessa cosa da soli, seguendo i suggerimenti dell'IDE, individuando quali sono i suggerimenti da applicare, fino a raggiungere l'abilità di apportare autonomamente tali modifiche.

Refactoring passo 2: modificare il codice della classe Mensola:

metodo setVolume

```
/**
 * Aggiunge un volume alla mensola
 * @param volume Volumer da aggiungere
 * @param posizione Posizione in cui aggiungere il volume
 * @return
 * @throws EccezionePosizioneOccupata La posizione in cui si cerca di aggiungere un volume è occupata, il volume non verrà aggiunto
 * @throws EccezionePosizioneNonValida La posizione non esiste
 */
public void setVolume(Libro volume, int posizione) throws EccezionePosizioneOccupata, EccezionePosizioneNonValida
{
    try
    {
        if (volumi[posizione]!=null)
            throw new EccezionePosizioneOccupata();
        volumi[posizione]=new Libro(1: volume); //aggiungo una COPIA di volume
       // return posizione;
    }
    catch(IndexOutOfBoundsException e)
    {
        //return -1;
        throw new EccezionePosizioneNonValida();
    }

}
```

Se la posizione non è vuota (è occupata) sollevo la relativa eccezione

Se la posizione non è valida (e quindi viene automaticamente sollevata l'eccezione ArrayIndexOutOfBoundException), sollevo la mia eccezione "Posizione non valida". Volendo si sarebbe potuto controllare la validità della posizione anche con gli "IF"

Le eccezioni sollevate vengono entrambe passate al "chiamante"

Si osservi che **non è più necessario che i metodi restituiscano un valore intero** poiché la gestione degli errori non avviene più con il codice (-1→ posizione non valida ecc..) ma con le eccezioni.

metodo getVolume

```
92  /**
93   * Restituisce il libro che si trova in una determinata posizione
94   * @param posizione  La posizione in cui cercare il libro
95   * @return
96   * @throws EccezionePosizioneVuota La posizione non contiene un volume
97   * @throws EccezionePosizioneNonValida La posizione non esiste
98   */
99  public Libro getVolume(int posizione) throws EccezionePosizioneVuota, EccezionePosizioneNonValida
00  {
01      try
02      {
03          return new Libro(volumi[posizione]);
04      }
05      catch(NullPointerException e)
06      {
07          throw new EccezionePosizioneVuota();
08      }
09      catch(ArrayIndexOutOfBoundsException e)
10      {
11          throw new EccezionePosizioneNonValida();
12      }
13
14  }
```

Se la posizione è vuota sollevo la relativa eccezione

Se la posizione non è valida (e quindi viene automaticamente sollevata l'eccezione ArrayIndexOutOfBoundException), sollevo la mia eccezione "Posizione non valida". Volendo si sarebbe potuto controllare la validità della posizione anche con gli "IF"

Le eccezioni sollevate vengono passate al "chiamante"

Metodo rimuoviVolume

```
/**
 * Libera (inserendo null) la posizione "posizione"
 * @param posizione La posizione da cui eliminare il volume
 * @return
 * @throws EccezionePosizioneNonValida La posizione non esiste
 * @throws EccezionePosizioneVuota La posizione non contien un volume
 */
public void rimuoviVolume(int posizione) throws EccezionePosizioneNonValida, EccezionePosizioneVuota
{
    /*
    if (posizione>=NUM_MAX_VOLUMI || posizione<0)
        return -1;
       */
     if (volumi[posizione]==null)
          // return -2;       //posizione vuota
           throw new EccezionePosizioneVuota();
    try
    {

        volumi[posizione]=null;
        //return posizione;
    }
    catch(ArrayIndexOutOfBoundsException e)
    {
        throw new EccezionePosizioneNonValida();
    }
}
```

Se la posizione è vuota sollevo la relativa eccezione

Se la posizione non è valida (e quindi viene automaticamente sollevata l'eccezione ArrayIndexOutOfBoundException), sollevo la mia eccezione "Eccezione non valida". Volendo si sarebbe potuto controllare la validità della posizione anche con gli "IF"

In tutti gli ulteriori metodi in cui si potrebbero verificare eccezioni (ad esempio nel costruttore di copia), si agisce passando tali eccezioni al chiamante.

Refactoring passo 3: modificare il codice della classe Scaffale:

Metodo setLibro

```java
/**
 * Inserisce il libro nella posizione "posizione" del ripiano "ripiano".
 * @param libro Il volume da inserire
 * @param ripiano Il ripiano dello scaffale in cui inserire il libro
 * @param posizione La posizione del ripiano in cui inserire il libro
 * @throws EccezionePosizioneOccupata  Se in quel ripiano-posizione è già contenuto un libro
 * @throws EccezionePosizioneNonValida Se la posizione non esiste
 * @throws EccezioneRipianoNonValido Se il ripiano non esiste
 */
//Il metodo può essere void poichè i vari casi sono gestiti con le eccezioni
public void setLibro(Libro libro, int ripiano, int posizione) throws EccezionePosizioneOccupata, EccezionePosizioneNonValida, EccezioneRipianoNonValido
{

    if (ripiano<0 || ripiano>=NUM_RIPIANI)
        throw new eccezioni.EccezioneRipianoNonValido();
    else
    {
        ripiani[ripiano].setVolume(volume: libro, posizione);
    }
}
```

Se li ripiano non è valido, sollevo la mia eccezione “Ripiano non valido”.

Le eccezioni sollevate, e quelle che “arrivano” dalla classe Mensola vengono tutte passate al “chiamante”

Metodo getLibro

```java
/**
 * Restituisce il volume nella posizione "posizione" del ripiano "ripiano".
 * @param ripiano Il ripiano da cui ottenere il volume
 * @param posizione La posizione, all'interno del ripiano, da cui ottenere il volume
 * @return
 * @throws EccezioneRipianoNonValido Se il ripiano non esiste
 * @throws EccezionePosizioneNonValida Se la posizione non esiste
 * @throws EccezionePosizioneVuota Se in quel ripiano-posizione non è presente un volume
 */
public Libro getLibro(int ripiano, int posizione) throws EccezioneRipianoNonValido, EccezionePosizioneNonValida, EccezionePosizioneVuota
{
    Libro libro;
    if (ripiano<0 || ripiano>=NUM_RIPIANI)
        throw new EccezioneRipianoNonValido();   //il ripiar
    libro=ripiani[ripiano].getVolume(posizione);
    return libro;
}
```

Se li ripiano non è valido, sollevo la mia eccezione “Ripiano non valido”.

Le eccezioni sollevate, e quelle che “arrivano” dalla classe Mensola vengono tutte passate al “chiamante”

Metodo rimuoviLibro

```
/**
 * Elimina il volume nella posizione "posizione" del ripiano "ripiano".
 * @param ripiano ripiano da cui rimuovere il volume
 * @param posizione posizione del ripiano da cuio rimuovere il volume
 * @throws EccezioneRipianoNonValido Se il ripiano non esiste
 * @throws EccezionePosizioneNonValida Se la posizione nonesiste
 * @throws EccezionePosizioneVuota se in quel ripiano-posizione non è presente il volume
 */
public void rimuoviLibro(int ripiano, int posizione) throws EccezioneRipianoNonValido, EccezionePosizioneNonValida, EccezionePosizioneVuota
{

    if (ripiano<0 || ripiano>=NUM_RIPIANI)
        throw new EccezioneRipianoNonValido();

    ripiani[ripiano].rimuoviVolume(posizione);
}
```

Se li ripiano non è valido, sollevo la mia eccezione “Ripiano non valido”.

Le eccezioni sollevate, e quelle che “arrivano” dalla classe Mensola vengono tutte passate al “chiamante”

metodo ElencotitoliAutore

```
public String[] elencoTitoliAutore(String autore) throws EccezioneRipianoNonValido, EccezionePosizioneNonValida
{
    Libro libro;
    String[] elencoTitoli;

    int contaLibriAutore=0;
    //conto il numero di libri di un autore presenti
    //per dimensionare l'array "elencoTitoli"
    for(int i=0;i<NUM_RIPIANI;i++)
    {
        for(int j=0;j<ripiani[i].getNumMaxVolumi();j++)
        {
            try
            {
                libro=getLibro(ripiano: i, posizione: j);
                if (libro.getAutore().equals(anObject: autore))
                    contaLibriAutore++;
            }
            catch (EccezionePosizioneVuota ex)
            {

            }
        }
    }
    if (contaLibriAutore==0)
        return null; //nessun libro dell'autore presente
```

Attenzione:

in questo caso, se durante il conteggio dei libri alcune posizioni sono vuote, in tal caso l’eccezione non deve fare terminare il metodo, quindi l’eccezione “Posizionevuota” viene gestita all’interno del metodo con un try-catch. La gestione consiste nel non fare assolutamente nulla, infatti, semplicemente, se una posizione è vuota , essa non viene conteggiata.

```
        elencoTitoli=new String[contaLibriAutore];  //istanzio l'array della dimensione giusta
        //Rufaccio il ciclo nidificato precedente per popolare l'array di titoli
        int posizioneTitolo=0;
         for(int i=0;i<NUM_RIPIANI;i++)
        {
            for(int j=0;j<ripiani[i].getNumMaxVolumi();j++)
            {
                try
                {
                    libro=getLibro(ripiano: i, posizione: j);
                    if (libro.getAutore().equals(anObject: autore))
                    {
                        elencoTitoli[posizioneTitolo]=libro.getTitolo();
                        posizioneTitolo++;
                    }
                }
                catch (EccezionePosizioneVuota ex)
                {

                }
            }
        }
         return elencoTitoli;   //restituisco l'array di titoli
    }
```

Caso analogo a quello visto sopra, se la postazione è vuota, il programma non fa nulla, procede.

metodo costruttore di copia Scaffale (Scaffale scaffale)

```
public Scaffale(Scaffale s) throws EccezioneRipianoNonValido, EccezionePosizioneNonValida, EccezionePosizioneOccupata
{
    Libro libro;
    //Istanzio l'array di mensole
    ripiani=new Mensola[s.getNumRipiani()];

    //Istanzio le 5 mensole
    // e in ogni mensola copio i libri

    for (int i=0;i<NUM_RIPIANI;i++)
    {
        ripiani[i]=new Mensola();
        {
            for (int j=0;j<ripiani[i].getNumMaxVolumi();j++)
            {
                try
                {
                    libro=s.getLibro(ripiano: i, posizione:j);
                    setLibro(libro, ripiano: i, posizione:j);
                }
                catch (EccezionePosizioneVuota ex)
                {
                   //non fare nulla...
                }

            }
        }
    }
}
```

Queste eccezioni vengono passate al chiamante ma comunque non si verificheranno mai se s1 è realizzato correttamente.

ATTENZIONE: poiché se la posizione è vuota, l'eccezione va gestita con un try-catch all'interno del metodo. Questo poiché altrimenti, se tale eccezione fosse passata al chiamante, quando viene rilevato scaffale/posizione vuoto, il metodo verrebbe interrotto non concludendo la costruzione dello scaffale di copia!

Se questa eccezione venisse passata al chiamante il costruttore di copia non funzionerebbe!!!

In tutti gli ulteriori metodi in cui si potrebbero verificare eccezioni si agisce passando tali eccezioni al chiamante.

Si osservi che **non è più necessario che i metodi restituiscano un valore intero** poiché la gestione degli errori non avviene più con il codice (-3→ ripiano non valido ecc..) ma con le eccezioni.

Refactoring passo 4: modificare il codice della classe App nelle parti in cui vengono invocati metodi della classe Scaffale. Si riporta solo la modifica alla voce del menu "Aggiungi volume", la gestione delle altre invocazioni dei metodi nella classe App è analoga:

```
case 2: // aggiungi volume
    libro=new Libro();
    System.out.println(x: "\nRipiano [0..4]--> ");
    ripiano=tastiera.nextInt();
    System.out.println(x: "\nPosizione [0..14]--> ");
    posizione=tastiera.nextInt();
    tastiera.nextLine();
    System.out.println(x: "\nTitolo--> ");
    libro.setTitolo(titolo:tastiera.nextLine());
    System.out.println(x: "\nAutore--> ");
    libro.setAutore(autore:tastiera.nextLine());
    System.out.println(x: "\nNumero pagine--> ");
    libro.setNumeroPagine(numeroPagine: tastiera.nextInt());
    try
    {
        s1.setLibro(libro, ripiano, posizione);
    }
    catch (EccezionePosizioneOccupata ex)
    {
        System.out.println(x: "Posizione non vuota");
    }
    catch (EccezionePosizioneNonValida ex)
    {
         System.out.println(x: "Posizione non valida");
    }
    catch (EccezioneRipianoNonValido ex)
    {
        System.out.println(x: "Ripiano non valido");
    }

    System.out.println(x: "Inserimento avvenuto correttamente");

    break;
```

Nel caso in cui si dovesse verificare un'eccezione, essa viene catturata e gestita con un try-catch. La gestione consiste semplicemente nel comunicare un messaggio di testo all'utente.

Esercizio 2:

Modificare l'esercizio GaraAutomobilistica consentendo la partecipazione ad un massimo di 10 partecipanti. Nel caso si tenti di aggiungere un partecipante oltre ai 10 si deve generare una eccezione "EccezionePostiDisponibiliEsauriti".