Alma Mater Studiorum-Università di Bologna
Scuola di Ingegneria

Eccezioni

Corso di Laurea in Ingegneria Informatica
Anno accademico 2021/2022

Prof. ENRICO DENTI
Dipartimento di Informatica – Scienza e Ingegneria (DISI)

SITUAZIONI CRITICHE

- In certe *situazioni critiche* è prevedibile che la *chiamata di un certo metodo* possa *causare errori*
  - l'apertura di un file potrebbe fallire
  - una connessione in rete potrebbe non riuscire
  - una conversione da stringa a numero potrebbe fallire
  - …
- Anche la *costruzione di un oggetto* potrebbe *risultare impossibile* a causa di *argomenti errati*
  - triangoli che non rispettano la condizione di esistenza
  - distanze negative
  - …

L' APPROCCIO “VECCHIA MANIERA”

- Alla «vecchia maniera», queste situazioni si gestiscono
  - inserendo nel metodo controlli per intercettare l'errore
  - inserendo nel chiamante controlli sul risultato (`null`? valido?)
- MA è un *approccio insoddisfacente,* perché:
  - non è facile prevedere tutte le configurazioni critiche
  - *il codice del chiamante diventa rumoroso*, pieno di `if`
  - “gestire” l’errore spesso significa solo stampare a video un messaggio.. che probabilmente non legge nessuno ☺ (server?)
  - ..senza realmente *propagare l'allarme* a chi di dovere
  - è del tutto *inutile* nel caso della *costruzione di oggetti,* un processo automatico che *non si può fermare!*

I NUOVI OBIETTIVI

- Separare nettamente la normalità dall'anomalia
  - *il task principale è fare le cose,* non gestire gli errori
    → *happy path* con business logic in evidenza
  - al contempo, evidenziare *quale sia e dove sia* il codice di gestione delle situazioni eccezionali
    → gestione errori su *error path* separato
  - *costrutto specifico* per error handling & recovery
  - riduzione/eliminazione oneri di controllo sul chiamante
- Allarme come entità che *interrompe* il normale flusso
  - in caso di errore, la priorità diventa *gestire l'emergenza*

IL CONCETTO DI ECCEZIONE

- Per questo, tutti i moderni linguaggi introducono la nozione di *eccezione*
  - un *allarme* lanciato da un'operazione quando, durante la sua esecuzione, si verifica una *situazione anomala*
  - l'allarme *non si stampa né si nasconde*: si gestisce
- Idea di fondo
  - le operazioni che *potrebbero* lanciare allarmi sono note
    - in alcuni linguaggi, come Java, sono *etichettate come tali* con un apposito tag nella signature (clausola **`throws`**)
  - devono quindi essere eseguite in un *ambiente controllato*
  - in caso di anomalia, il controllo passa a un *gestore di eccezione* che si fa carico di gestire il problema

IL MECCANISMO `try/catch`

- L'operazione che *potrebbe* lanciare un allarme è *eseguita all'interno di un blocco controllato*
  - si tenta (*try*) di eseguire l'operazione: se qualcosa va storto, *l'allarme parte automaticamente*
    - viene "*lanciata l'eccezione*"
  - In tal caso *non si prosegue normalmente:* l'allarme viene *catturato* dal gestore (*catch*) e l'esecuzione continua lì

```
try {
    // operazione critica
}
catch (tipoeccezione e) {
    // in caso di problemi
    // si continua qui
}
// se tutto va bene
// si prosegue qui
```

Java | C# | ~Scala | Kotlin

NB: in Scala, la sintassi del catch è leggermente diversa

IL MECCANISMO `try/catch`

- Per distinguere diversi tipi di problemi, *un blocco* ***`try`*** *può essere seguito da più blocchi* ***`catch`***
  - se qualcosa va storto, *parte l'allarme appropriato*
    - viene "lanciata l'eccezione" *del tipo opportuno*
  - si prosegue nel *gestore specifico* per quel tipo di allarme
    - in caso di *tipoeccezione1* si continua nel primo
    - altrimenti, nel secondo
    - … e così via

NB: in Scala vi è invece un catch unico, con dentro i diversi *case*

```
try {
   op1();
   dangerousOp(..);
   op2();
}
catch (tipoeccezione1 e1){
   ...
}
catch (tipoeccezione2 e2){
   ...
}
...
```

Java C# ~Scala Kotlin

IL MECCANISMO `try/catch`

- Eventuali azioni da svolgere in ogni caso alla fine possono essere specificate in *un blocco* ***`finally`***
  - abbastanza raro
  - specifica le azioni da eseguirsi *sia che l'operazione sia stata eseguita normalmente, sia che sia stata lanciata eccezione*
  - usato per rilasciare le risorse che l'operazione interrotta stava usando.

NB: in Scala vi è invece un catch unico, con dentro i diversi *case*

```
try {
   ...
}
catch (tipoeccezione1 e1){
   ...
}
catch (tipoeccezione2 e2){
   ...
}
finally {
   ...
}
```

Java C# ~Scala Kotlin

DIVERSI LIVELLI DI “RISCHIOSITÀ”

- Alcune operazioni sono *potenzialmente «spesso» rischiose*
  - ESEMPIO: accesso al «mondo esterno»
    - apertura di file → rischio file non trovato
    - apertura connessione di rete → rischio rete assente
- Altre invece *potrebbero esserlo o no,* a seconda del contesto
  - ESEMPIO: conversione stringa / numero
    - se le stringhe da convertire provengono da file, potrebbero contenere errori → *l'operazione potrebbe essere rischiosa*
    - se, invece, provengono da fonte certificata, la conversione non può fallire → *l'operazione non presenta rischi*

OBBLIGHI DIVERSI PER SITUAZIONI DIVERSE…?

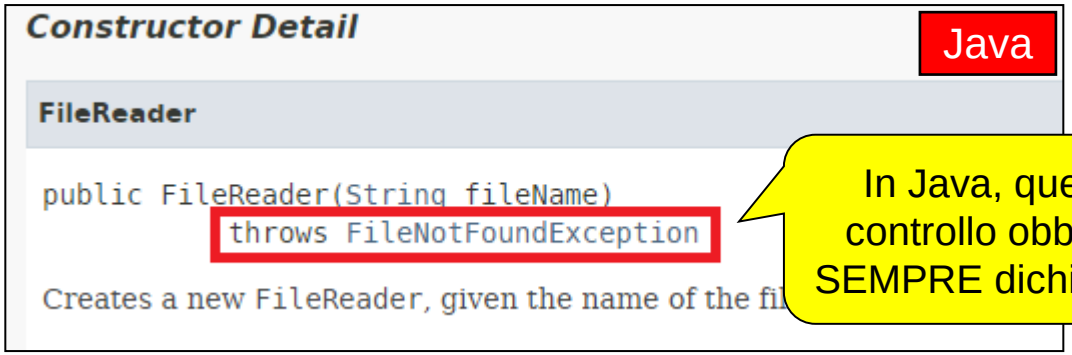
- Alcuni linguaggi scelgono perciò di *differenziare tali scenari,* distinguendo fra: Java
  - eccezioni a controllo obbligatorio (*checked exceptions*)
    → uso di **`try/catch`** *obbligatorio*
  - eccezioni a controllo facoltativo (*unchecked exceptions*)
    → uso di **`try/catch`** *facoltativo*
- Altri invece scelgono di *non farlo* C# Scala Kotlin
  - solo eccezioni a controllo facoltativo (*unchecked exceptions*)
    → uso di **`try/catch`** *sempre facoltativo*
- MOTIVO:
  - l'esperienza su piccola e media scala indicava l'opportunità del controllo obbligatorio, quella su larga scala invece…

OBBLIGHI DIVERSI PER SITUAZIONI DIVERSE…?

- Checked exceptions: perché sì
  - obbligano il chiamante a considerare il caso eccezionale
  - funzionano bene per sistemi auto-contenuti, in cui la "distanza" fra chi lancia l'eccezione e chi la cattura è "piccola"
- Checked exceptions: perché no
  - non scalano bene all'aumentare della complessità
    - in un sistema a molti livelli, i livelli alti operano a un livello di astrazione tale da non voler / poter sapere cosa accade sotto
    - se un tipico sotto-sistema lancia 4-10 eccezioni, aggregandoli si può arrivare a doverne gestire decine, anche 80+ → *insostenibile*
  - ostacolano lo sviluppo di nuove versioni
    - se una nuova versione aggiunge una nuova eccezione da gestire *e questa va dichiarata perché è a controllo obbligatorio*, si rompe la retro-compatibilità col codice pre-esistente

ESEMPIO 1: APERTURA DI UN FILE

- Come vedremo, in Java l'apertura di un file di testo comporta la costruzione di un oggetto **`FileReader`**
  - in C#, **`StreamReader`**, con piccole differenze sintattiche
- L'operazione è critica perché il file *potrebbe non esistere*
  - in tal caso, il costruttore lancia una **`FileNotFoundException`**



ESEMPIO 1: APERTURA DI UN FILE

- Come vedremo, in Java l'apertura di un file di testo comporta la costruzione di un oggetto **`FileReader`**
  - in C#, **`StreamReader`**, con piccole differenze sintattiche
- L'operazione è critica perché il file *potrebbe non esistere*
  - in tal caso, il costruttore lancia una **`FileNotFoundException`**
- In Java, ciò implica controllo obbligatorio: l'apertura del file deve *necessariamente* avvenire in un *blocco* ***`try/catch`***

Java

```
try {
  FileReader f = new FileReader("info.txt");
}
catch(FileNotFoundException e){
   // gestione eccezione
}
```

ESEMPIO 1 con `try/catch`

- Se il file manca, il normale flusso di esecuzione *si interrompe* e il controllo passa al *gestore di eccezione*
  - in questo caso, la "gestione" consiste semplicemente nell'emettere un messaggio e forzare la terminazione pulita

Java

```
class Esempio1 {
 public static void main(String[] args){
  try { FileReader f = new FileReader(args[0]); }
  catch(FileNotFoundException e){
   System.err.println("File " + args[0] + " not found");
   System.exit(1); // esce con indicazione d'errore
  }
 }
}
```

ESEMPIO DI ESECUZIONE (CASO FILE ASSENTE)

```
> java Esempio2 info.txt
File info.txt not found
```

GESTIRE = … *TERMINARE ?*

NO!

- È improbabile che la scelta migliore sia forzare la terminazione: *spesso la situazione non è così grave*
  - se quel file era essenziale, forse un abort ha anche senso…
  - ..ma se serviva solo per caricare preferenze, una immagine, dei colori.. si può far senza!
    - impostazioni di default
    - ignorare la cosa e buona notte…
- Exit o Abort a tappeto??
  No, grazie! ☺

ESEMPIO 1 senza `try/catch`

- Essendo questa eccezione a controllo obbligatorio, se il **`try` / `catch`** manca Java dà *errore di compilazione*

Java

```
class Esempio1 {
 public static void main(String[] args){
  FileReader f = new FileReader(args[0]); // errore
 }
}
```

```
Esempio1.java:4:
unreported exception java.io.FileNotFoundException;
must be caught or declared to be thrown
       FileReader f = new FileReader(args[0]);
                                     ^1 error
```

ESEMPIO 1 in C# (senza `try/catch`)

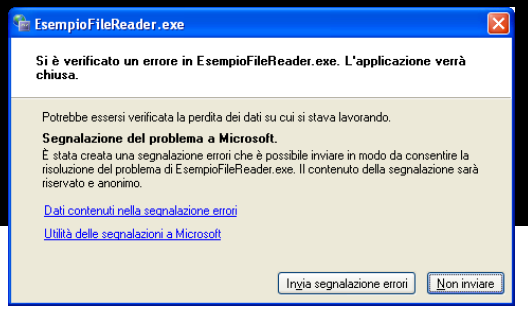
- In C# invece il controllo delle eccezioni è sempre opzionale, quindi questo programma è accettato e compilato:

C#

```
using System.IO;
public class Esempio1 {
 public static void Main(string[] args){
    StreamReader f = File.OpenText(args[0]);
 }
}
```

MA se a runtime il file risulta mancante, accade il disastro: il programma abortisce

```
Unhandled Exception: System.IO.FileNotFoundException:
Could not find file 'pippo'
at System.IO.File.OpenText(String path)
   at EsempioReader.Main(String[] args)
```



ESEMPIO 1 in Scala

- In Scala il **`catch`** non ha argomenti, perché i tipi di eccezioni sono specificati come ***`case`*** all'interno

Scala

```
object Esempio2 {
 def main(args: Array[String]) : Unit = {
     try {
       val source = scala.io.Source.fromFile(args(0))
     }
     catch {
       case e: java.io.FileNotFoundException => {
           println("File " + args(0) + " non trovato");
           sys.exit(1);
       }
     }
 }
}
```

Scala riutilizza molte eccezioni Java, senza neppure ridefinirle

Eventuali catch multipli si esprimono come ***`case`*** *multipli* dentro a un *unico* blocco **`catch`**

ESEMPIO 1 in Kotlin

- In Kotlin il costrutto è molto simile a Java

Kotlin

```
fun main(args: Array<String>) {
  try {
     val f = java.io.File(args[0]).reader();
     }
  catch(e: java.io.FileNotFoundException){
     println("File " + args[0] + " non trovato");
     kotlin.system.exitProcess(1);
  }
  catch(e: Exception){
     println(e);
     kotlin.system.exitProcess(2);
  }
}
```

Anche Kotlin riutilizza molte eccezioni Java senza neppure ridefinirle

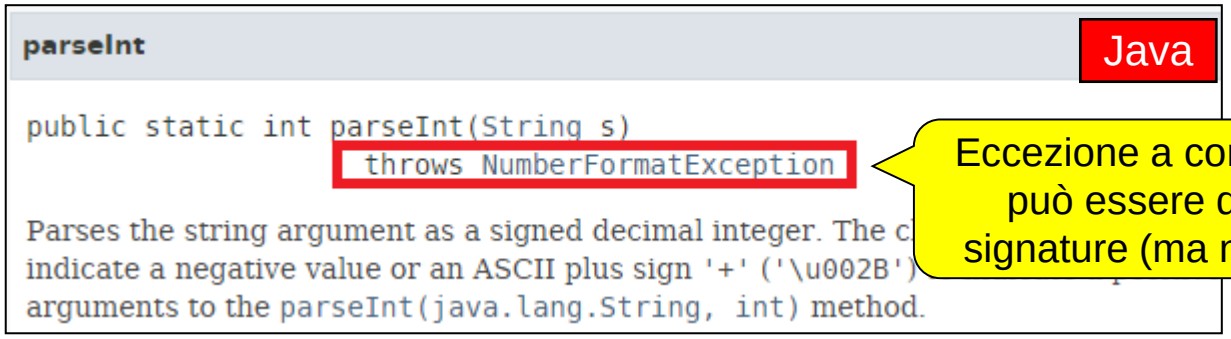
ESEMPIO 2: CONVERSIONE STRINGA / NUMERO

- La conversione stringa / intero è svolta in Java dalla funzione statica di libreria

  `int Integer.parseInt(String s)`

  - in C#, analoga funzionalità è fornita da **`int.Parse(string s)`**

- L'operazione è *potenzialmente* critica, perché la stringa *potrebbe* non contenere la rappresentazione di un intero
  - in tal caso viene lanciata una **`NumberFormatException`**



ESEMPIO 2: CONVERSIONE STRINGA / NUMERO

- La conversione stringa / intero è svolta in Java dalla funzione

  `int Integer.parseInt(String s)`

  - in C#, analoga funzionalità è fornita da **`int.Parse(string s)`**
- L'operazione è *potenzialmente* critica, perché la stringa *potrebbe* non contenere la rappresentazione di un intero
  - in tal caso viene lanciata una **`NumberFormatException`**
- A differenza del caso precedente, però, la chiamata a **`parseInt`** non è *necessariamente* a rischio
  - se manipola dati noti e già verificati, l'operazione è *sicura a priori*
- Perciò, in Java questo **`try/catch`** *non è obbligatorio* perché in molti casi costituirebbe un appesantimento inutile.

CON O SENZA `try/catch` ?

- Per le eccezioni a controllo facoltativo, omettere il blocco **`try`** / **`catch`** *non causa* la protesta del compilatore…
- …MA se poi i dati non sono verificati e contengono errori, si genera il *disastro a runtime*
  - se una stringa non è un numero, parte l'eccezione
  - se nessuno la cattura, *essa "uccide il main" come un proiettile,* e il programma abortisce
- Può quindi aver senso prevedere la situazione e catturare l'errore, *per salvare il salvabile*
  - l'operazione col dato errato è "condannata"..
  - ..ma *gestire l'eccezione permette di proseguire con le successive,* senza causare l'abort di tutto il programma per un dato errato

ESEMPIO 2 con `try/catch`

- Se una stringa *non è un numero*, parte l'eccezione
- Catturando l'errore, il gestore *permette di proseguire con le operazioni successive*

Java

```
public static void main(String args[]){
   String[] stringhe = {"-12", "112a", "14" };

   for (String s : stringhe) {
    try {
     int a = Integer.parseInt(s);
     System.out.println("Stringa ok: " + s);
    }
    catch (NumberFormatException e) {
     System.out.println("Stringa errata" + s);
    }
  }
}
```

```
Stringa ok: -12
Stringa errata: 112a
Stringa ok: 14
```

ESEMPIO 2 senza `try/catch`

- Non catturando l'errore, il primo dato errato causa l'abort

Java

```
class Esempio2 {
 public static void main(String args[]){
   String[] stringhe = {"-12", "112a", "14" };
    for (String s : stringhe) {
     int a = Integer.parseInt(s);
     System.out.println("Stringa ok: " + s);
   }
 }
}
```

Senza try/catch, alla 2ª iterazione l'eccezione "colpisce al cuore" il main: il "14" non sarà mai elaborato, anche se era corretto!

```
Stringa ok: -12
Exception in thread "main" java.lang.NumberFormatException: For input string:
"112a"
 at java.lang.NumberFormatException.forInputString(Unknown Source)
 at java.lang.Integer.parseInt(Unknown Source)
 at java.lang.Integer.parseInt(Unknown Source)
 at EsempioEccezioneSenzaTryCatch.main(EsempioSenzaTryCatch.java:5)
```

ESEMPIO 2 in Scala

Scala

```
object EsempioEccezioneConTry {
 def main(args: Array[String]) : Unit = {
   val stringhe = Array("-12", "112a", "a12");
   for (s <- stringhe) {
    try {
     val a : Int = s.toInt;
     println("Stringa ok: " + s);
    }
    catch {
       case e: java.lang.NumberFormatException =>
                      println("Stringa errata: " + s);
    }
   }
 }
}
```

ESEMPIO 2 in Kotlin

Kotlin

```
fun main(args: Array<String>) {
   val stringhe = arrayOf("-12", "112a", "a12");
   for (s in stringhe) {
    try {
     val a : Int = s.toInt();
     println("Stringa ok: " + s);
    }
    catch (e: java.lang.NumberFormatException) {
     println("Stringa errata: " + s);
    }
   }
}
```

ECCEZIONI DI DIVERSA NATURA

- Possibili linee guida
  - per le eccezioni che derivano da situazioni *esterne*, fuori dal nostro controllo, può aver senso pretendere una politica di gestione per far fronte al problema → *try/catch obbligatorio*
    - non è un bug se a runtime il file manca o la rete è giù!
  - per quelle che invece hanno una *sorgente interna,* ossia dipendono da un nostro bug, non ha senso pretenderlo, perché se è un nostro bug dovevamo risolverlo → *try/catch facoltativo*
    - rappresentano *violazioni di precondizioni:* se il cliente ha fornito un dato assurdo, cosa possiamo farci? Mica inventarne un altro!
    - solitamente a runtime la situazione non è recuperabile, se non per "salvare il salvabile" per le operazioni *successive*

CONTROLLO OBBLIGATORIO vs CONTROLLO FACOLTATIVO

- In Java

  Java

  – *si distingue* fra *checked* e *unchecked* exceptions
  – le eccezioni che rappresentano situazioni *fuori dal nostro controllo* derivano da **`Exception`** → controllo *obbligatorio*
  – le eccezioni che rappresentano *violazioni di precondizioni* derivano invece dalla sottoclasse **`RuntimeException`** → controllo *facoltativo*

- In C#, Scala, Kotlin

  C# Scala Kotlin

  – *non si distingue* fra *checked* e *unchecked* exceptions
  – tutte le eccezioni sono *unchecked* → controllo *sempre facoltativo*
  – sta al progettista valutare quando, cosa e dove controllare
  – però, il fatto che di base siano tutte unchecked *non è una scusa per abdicare al proprio ruolo:* è un'altra *dimensione del progetto*

NEGLI ESEMPI PRECEDENTI…

- ESEMPIO 1
  La mancanza di un file a runtime *non è un nostro errore di programmazione*: è una situazione fuori dal nostro controllo
  - per questo, la **`FileNotFoundException`** lanciata dal costruttore di **`FileReader`** è, in Java, a controllo *obbligatorio*
- ESEMPIO 2
  Convertire in numero una stringa con caratteri non numerici è probabilmente la spia di *un errore di programmazione*
  - il cliente che chiama **`parseInt`** con quell'argomento viola il contratto d'uso, ma a runtime ormai ci si può fare ben poco
    → controllo *facoltativo*
  - se presente, il controllo mira alla *riduzione del danno*

MECCANISMO vs SCELTE DI PROGETTO

- Un'eccezione *a controllo obbligatorio* non lascia margini
- Un'eccezione *a controllo facoltativo* lascia margini di scelta
  - si può decidere di controllarle comunque ("per prudenza")
  - si può optare invece per non controllarle (quasi) mai
- Pro & contro
  - l'approccio "prudente" previene qualunque rischio, *ma al prezzo di complicare il codice al punto di renderlo talora quasi illeggibile*
    - possono essere più i try/catch della "business logic"
  - l'approccio "vada come vada" *mantiene il codice più snello, ma accetta intrinsecamente dei rischi*
    - cruciale intercettare i bug a tempo di sviluppo → testing
  - PRASSI: linee guida "team-based" con *sensate "vie di mezzo"*

SCENARI TIPICI CON FORMATTATORI NUMERICI E DATE

- I formattatori di valori numerici ottenuti da **`NumberFormat`**, a differenza di **`parseInt`**, lanciano eccezioni *a controllo obbligatorio* nei loro metodi **`parse`**
  - motivo: nel 99% dei casi, quelle stringhe provengono dall'esterno, quindi, prevale la prudenza
  - RICORDA: nell'ambiente **`jshell`** si può evitare di controllare le eccezioni a controllo obbligatorio, perché l'ambiente interattivo è fatto per sperimentare proprio le singole istruzioni

```
jshell> import java.text.NumberFormat;

jshell> NumberFormat f = NumberFormat.getNumberInstance();
f ==> java.text.DecimalFormat@674dc

jshell> f.parse("1.300,25");
$3 ==> 1300.25

jshell> f.parse("1300,25");
$4 ==> 1300.25
```

Java

Il Locale di default è Italy

Si aspetta valori reali con la virgola come separatore decimale ed eventualmente il punto come separatore delle migliaia

SCENARI TIPICI CON FORMATTATORI NUMERICI E DATE

- I formattatori di valori numerici ottenuti da **`NumberFormat`**, a differenza di **`parseInt`**, lanciano eccezioni *a controllo obbligatorio* nei loro metodi **`parse`**
  - in un programma, l'eccezione va necessariamente controllata
  - il metodo **`parse`** restituisce un oggetto **`Number`** – una classe di utilità che incapsula un valore numerico primitivo, di vario tipo
  - il metodo **`parse`** può lanciare una **`ParseException`**

Java

```
NumberFormat f = NumberFormat.getNumberInstance();
try {
    Number n = f.parse("10,50");
    System.out.println(n);
}
catch(ParseException e){
    System.out.println("parsing fallito");
}
```

Il Locale di default è Italy, quindi si aspetta valori reali con la virgola

SCENARI TIPICI CON FORMATTATORI NUMERICI E DATE

- Ci si aspetterebbe quindi che il metodo **`parse`** lanci eccezione appena trova una stringa «insolita», *ma così non è*
  - motivo (molto discutibile): limitare i casi in cui lanciare «davvero» eccezione a quelli «certamente» sbagliati
    → MA: chi decide quali sono i casi «veramente sbagliati»?
  - RISCHIO: far passare cose in realtà errate e, peggio, *restituire in tal caso valori insensati*
    → E infatti…

```
jshell> f.parse("1.300,25");
$3 ==> 1300.25

jshell> f.parse("1300,25");
$4 ==> 1300.25

jshell> f.parse("1300.25");
$5 ==> 130025

jshell> f.parse("1300-25");
$6 ==> 1300

jshell> f.parse("1300+25");
$7 ==> 1300

jshell> f.parse("1300a25");
$8 ==> 1300

jshell> f.parse("1300:25");
$9 ==> 1300
```

Ignora il separatore delle migliaia sempre e comunque

Inoltre, se dopo un numero corretto c'è qualcosa di scorretto, si ferma alla parte corretta ignorando l'altra

SCENARI TIPICI CON FORMATTATORI NUMERICI E DATE

- Ci si aspetterebbe quindi che il metodo **`parse`** lanci eccezione appena trova una stringa «insolita», *ma così non è*
  - solo se la stringa *non inizia con un numero accettabile* lancia effettivamente eccezione

Inizia con un carattere non numerico
→ ce la dà su

```
jshell> f.parse(":1300,25");
  Exception java.text.ParseException: Unparseable number: ":1300,25"
        at NumberFormat.parse (NumberFormat.java:434)
        at (#10:1)
```

  - motivo: semplificare la realizzazione di parser che debbano estrarre una parte numerica da una frase più lunga
  - MA: così diventa impossibile validare davvero l'input!

SCENARI TIPICI CON FORMATTATORI NUMERICI E DATE

- Il **`Locale`** selezionato e, nel caso di date e orari, il tipo di formato (**`Short`**, **`Medium`**, **`Long`**) determinano l'insieme delle stringhe ammissibili

```
jshell> NumberFormat f = NumberFormat.getNumberInstance(Locale.US);
f ==> java.text.DecimalFormat@674dc

jshell> f.parse("1300.25");
$21 ==> 1300.25

jshell> f.parse("1,300.25");
$23 ==> 1300.25

jshell> f.parse("1.300,25");
$24 ==> 1.3
```

Locale USA

Nel formato USA, il punto è separatore decimale: la successive virgola è *illegale in quella posizione* e quindi il parsing ignora ciò che segue

```
jshell> NumberFormat f = NumberFormat.getNumberInstance(Locale.CANADA_FRENCH);
f ==> java.text.DecimalFormat@674dc

jshell> f.parse("1300.25");
$15 ==> 1300

jshell> f.parse("1300,25");
$16 ==> 1300.25

jshell> f.parse("1.300,25");
$17 ==> 1

jshell> f.parse("1 300,25");
$18 ==> 1

jshell> f.parse("1\u00A0300,25");
$19 ==> 1300.25
```

Locale CANADA_FRENCH

Il separatore delle migliaia corretto è lo spazio hard, '\u00A0'

SCENARI TIPICI CON FORMATTATORI NUMERICI E DATE

- Il **`Locale`** selezionato e, nel caso di date e orari, il tipo di formato (**`Short`**, **`Medium`**, **`Long`**) determinano l'insieme delle stringhe ammissibili

```
jshell> DateTimeFormatter f = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.SHORT);
f ==> Localized(SHORT,SHORT)

jshell> f.parse("26/03/22, 15:30");
$45 ==> {},ISO resolved to 2022-03-26T15:30

jshell> f.withLocale(Locale.US).parse("3/26/22, 3:30 PM");
$46 ==> {},ISO resolved to 2022-03-26T15:30
```

Locale USA

```
jshell> f.withLocale(Locale.CANADA_FRENCH).parse("2022-03-26 15 h 30");
$53 ==> {},ISO resolved to 2022-03-26T15:30

jshell> f.withLocale(Locale.CANADA_FRENCH).parse("2022-03-26\u00A015\u00A0h\u00A030");
|  Exception java.time.format.DateTimeParseException: Text '2022-03-26 15 h 30' could no
|        at DateTimeFormatter.parseResolved0 (DateTimeFormatter.java:2052)
|        at DateTimeFormatter.parse (DateTimeFormatter.java:1880)
|        at (#54:1)
```

Locale CANADA_FRENCH

Nel parsing *lo spazio atteso è quello classico*, NON quello hard '\u00A0'

SCENARI TIPICI CON FORMATTATORI NUMERICI E DATE

- Il **`Locale`** selezionato e, nel caso di date e orari, il tipo di formato (**`Short`**, **`Medium`**, **`Long`**) determinano l'insieme delle stringhe ammissibili

```
jshell> DateTimeFormatter f = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.MEDIUM);
f ==> Localized(MEDIUM,MEDIUM)

jshell> f.parse("26 mar 2022, 15:30");
|  Exception java.time.format.DateTimeParseException: Text '26 mar 2022, 15:30' could not be
|        at DateTimeFormatter.parseResolved0 (DateTimeFormatter.java:2052)
|        at DateTimeFormatter.parse (DateTimeFormatter.java:1880)
|        at (#57:1)

jshell> f.parse("26 mar 2022, 15:30:00");
$58 ==> {},ISO resolved to 2022-03-26T15:30
```

Locale ITALY

Nel formato MEDIUM, *i secondi sono obbligatori*

```
jshell> DateTimeFormatter f = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.LONG, FormatStyle.SHORT);
f ==> Localized(LONG,SHORT)

jshell> f.parse("26 marzo 2022, 15:30");
$61 ==> {},ISO resolved to 2022-03-26T15:30
```

Nel formato MEDIUM + SHORT, no

SCENARI TIPICI CON FORMATTATORI NUMERICI E DATE

- Spesso questo comportamento NON è ciò che vorremmo
  - per validare l'input serve che **`parse`** avvisi se c'è qualche carattere illegale nella stringa, *non che si fermi e faccia finta di niente!*
- Si può ovviare con il metodo **`parse`** a due argomenti
  - questo metodo NON lancia mai eccezione: lascia all'utente la scelta
  - il secondo argomento, una **`ParsePosition`**, dice "fin dove è arrivato" il parsing → *basta controllare se è arrivato in fondo*

```
ParsePosition position = new ParsePosition(0);
Number n = f.parse("10-10", position);
if (position.getIndex() != str.length()) {
 … // decidere cosa fare
}
```

Java

Così, se il parsing non arriva fino in fondo, ce ne accorgiamo e provvediamo noi

SCENARI TIPICI CON FORMATTATORI NUMERICI E DATE

- Esempio
  - NB: la **`ParsePosition`** va resettata a ogni chiamata di **`parse`**, o costruendo un nuovo oggetto o invocando **`setIndex`**

```
jshell> import java.text.ParsePosition

jshell> pos = new ParsePosition(0);
pos ==> java.text.ParsePosition[index=0,errorIndex=-1]

jshell> f.parse("1.300,25", pos);
$71 ==> 1300.25

jshell> pos
pos ==> java.text.ParsePosition[index=8,errorIndex=-1]

jshell> pos.setIndex(0)

jshell> f.parse("1.300+25", pos);
$74 ==> 1300

jshell> pos
pos ==> java.text.ParsePosition[index=5,errorIndex=-1]
```

Java

Reset della ParsePosition prima di una nuova **`parse`**

Parsing giunto solo alla posizione 5 (il '+', escluso)

SCENARI TIPICI CON FORMATTATORI NUMERICI E DATE

- MA occhio al separatore delle migliaia!
  - esso viene comunque ignorato, quindi in tal caso **`ParsePosition`** arriva comunque in fondo ☹ → necessario un trattamento ad hoc

Separatore delle migliaia usato erroneamente

Separatore delle migliaia usato erroneamente

Java

```
jshell> f.parse("1300.25", pos);
$77 ==> 130025

jshell> pos.setIndex(0)

jshell> f.parse("1.300.25", pos);
$79 ==> 130025

jshell> pos
pos ==> java.text.ParsePosition[index=8,errorIndex=-1]
```

Parsing *giunto comunque fino in fondo*, perché ha ignorato il separatore delle migliaia!

  - si può recuperare tale separatore per il **`Locale`** corrente invocando i metodi **`getDecimalFormatSymbols`** e **`getGroupingSeparator`**

    NB: occorre un cast perché il formattatore restituito dal metodo factory di **`NumberFormat`** è in realtà un **`DecimalFormat`**

```
jshell> var sepMigliaia = ((DecimalFormat)f).getDecimalFormatSymbols().getGroupingSeparator()
sepMigliaia ==> '.'
```

SCENARI TIPICI CON FORMATTATORI NUMERICI E DATE

- Anche i formattatori di date lanciano eccezioni *a controllo obbligatorio* nei loro metodi **`parse`**
  - formato e Locale determinano l'insieme delle stringhe ammissibili
- Occhio, però, allo schema d'uso!
  - oltre che chiedere al formattatore di fare il parsing, lo si può chiedere all'oggetto LocalDateTime, passandogli il formattatore

Java

```
DateTimeFormatter formatter =
        DateTimeFormatter.ofLocalDateTime(..);
LocalDateTime myDateTime =
      LocalDateTime.parse("13/01/2012 20.04", formatter);

ZonedDateTime apollo11Launch =
    ZonedDateTime.parse(
      "1969-07-16T03:32-04:00[America/New_York]",
      DateTimeFormatter.ISO_DATE_TIME);
```

Usa un nostro formattatore

Usa uno dei formattatori standard ISO

Lancio deliberato di eccezioni

CHI PUÒ LANCIARE ECCEZIONI?

- *Molte operazione di libreria* lanciano eccezioni
  - la documentazione specifica per ogni operazione *quale eccezione* possa essere lanciata
  - in Java → dichiarazione **`throws`** nella signature
- *Noi stessi possiamo farlo* in presenza di una situazione ritenuta per qualche motivo *"critica"*
  - per *impedire la costruzione* di *oggetti inconsistenti*
  - per *interrompere l'esecuzione* di un metodo che si trovi impossibilitato a proseguire *con coerenza*
  - per emettere all'esterno un allarme *di tipo specifico* ritenuto *più adatto a descrivere l'accaduto*
  - in Java → dichiarazione **`throws`** nella signature
  - istruzione **`throw`** per «lanciare» materialmente l'eccezione

LANCIO DELIBERATO DI ECCEZIONI

- Ha senso *lanciare deliberatamente eccezione* se un nostro metodo incontra una *situazione inconsistente*
  - un metodo richiede un argomento *non negativo*, ma gli viene passato -5
  - il costruttore si accorge che, con gli argomenti dati, risulta *impossibile costruire un oggetto sensato*
- A tal fine:
  1. si crea esplicitamente un *oggetto-eccezione* adatto
  2. lo si lancia mediante *l'apposita istruzione* **`throw`**
  3. per eccezioni a controllo obbligatorio, in Java la signature del metodo deve contenere la *dichiarazione* **`throws`**

SCENARI TIPICI

- In **`Frazione`**, il costruttore richiede denominatore ≠ 0
  - se gli si passa 0, si viola il contratto d'uso (precondizione)..
    .. *ma il povero costruttore di* `Frazione` *cosa può farci?*
- In **`Triangolo`**, il costruttore richiede il rispetto della condizione di esistenza
  - passargli tre valori che non la rispettano è una chiara violazione di precondizione, ma.. *il costruttore di* `Triangolo` *cosa può fare?*
  - *se anche verifica gli argomenti, ormai è tardi: la costruzione è già in corso e non si può più fermare!*
- Un metodo che sorteggi un *numero del lotto* dovrebbe fornire valori nell'intervallo [1..90]
  - .. ma se per errore viene generato un valore diverso?

UN APPROCCIO MIGLIORE

- In questi casi, la via giusta è *lanciare un'eccezione*
  - il corpo del metodo **deve** creare l'eccezione e lanciarla

    `throw new ExceptionType(…)`
  - in Java, la signature del metodo **può** includere la dichiarazione

    `throws ExceptionType`

    (normalmente solo per eccezioni a controllo obbligatorio)
- Il *tipo* dell'eccezione deve *indicare cosa è successo*
  - *per argomenti errati*, **`IllegalArgumentException`** (sottoclasse di **`RuntimeException`**) o sue classi derivate
  - *per altri problemi*, una qualche sottoclasse di **`Exception`** (scelta in base alla situazione specifica)

SCENARIO TIPICO: `Frazione`

- Il costruttore di **`Frazione`** può *lanciare un'opportuna eccezione* se rileva che il denominatore è 0
  - il messaggio d'errore incapsulato deve spiegare l'accaduto
  - per violazioni di precondizioni, l'eccezione scelta è praticamente sempre una **`IllegalArgumentException`** (IAE)

Java ~C#

```
class Frazione {
 private int num, den;
 public Frazione(int num, int den){
  if (den==0) throw new IllegalArgumentException("den is zero");
  if (den>0) {this.num=num; this.den=den;}
  else { this.num=-num; this.den=-den;}
 }
 public String toString(){
  return String.valueOf(num) + "/" + String.valueOf(den);
 }
}
```

SCENARIO TIPICO: `Frazione`

- Il costruttore di **`Frazione`** può *lanciare un'opportuna eccezione* se rileva che il denominatore è 0
  - il messaggio d'errore incapsulato deve spiegare l'accaduto
  - per violazioni di precondizioni, l'eccezione scelta è praticamente sempre una **`IllegalArgumentException`** (IAE)

```
jshell> var f1 = new Frazione(2,5);
f1 ==> 2/5

jshell> var f2 = new Frazione(2,0);
|  Exception java.lang.IllegalArgumentException: den is zero
|        at Frazione.<init> (#85:4)
|        at do_it$Aux (#87:1)
|        at (#87:1)
```

In caso di precondizioni violate, viene lanciata l'eccezione del tipo prescelto, che incapsula il messaggio d'errore specificato

DICHIARAZIONE `throws`: MECCANISMO & PRASSI

- La dichiarazione **`throws`** nella signature in Java è
  - obbligatoria per le eccezioni a controllo obbligatorio
  - facoltativa per le eccezioni a controllo facoltativo
- Quando metterla?
  - *sempre e comunque* sarebbe inutilmente pesante
  - prassi: *metterla solo se rappresenta qualcosa di "insolito"*
    - per coerenza con la natura "facoltativa" della gestione normale
    - per limitare l'elenco ai casi "realmente rilevanti", quelli in cui *non ce lo si aspetterebbe*

Nei prossimi esempi evidenzieremo le due situazioni.

DICHIARAZIONE `throws`: SCENARIO TIPICO `Frazione`

- Nei costruttori:
  - è normale verificare le precondizioni (per **`Frazione`**, che il denominatore non sia zero)
  - per violazioni di precondizioni, l'eccezione scelta è praticamente sempre una **`IllegalArgumentException`** (IAE)
- Quindi, è inutile e sovrabbondante specificarla

```
class Frazione {
 private int num, den;
 public Frazione(int num, int den){
  if (den==0) throw new IllegalArgumentException("den is zero");
  if (den>0) {this.num=num; this.den=den;}
  else { this.num=-num; this.den=-den;}
 }
 ...
}
```

Dichiarazione throws ASSENTE perché scontata

Java

~C#

DICHIARAZIONE `throws` : SCENARIO TIPICO `Triagolo`

- Analogamente accade in **`Triangolo`**

```
public Triangolo(double a, double b, double c)
        throws IllegalArgumentException {

   if (condizione di esistenza violata) {
     throw new IllegalArgumentException(…);
   }
   // happy path: inizializza normalmente
   this.a=a; this.b=b; this.c=c;
}
```

Java

~C#

Java: comunque non indispensabile
Per prassi si omette

Lanciando eccezione se gli argomenti violano il contratto, il costruttore:

- interrompe le costruzione di un oggetto certamente inconsistente
- segnala al cliente la situazione eccezionale verificatasi

ESEMPIO: TRIANGOLO IMPOSSIBILE

In Scala e Kotlin, il costruttore primario è «built-in» nell'intestazione:

- in Scala, la parte di verifica si aggiunge di seguito
- In Kotlin, la parte di verifica si aggiunge in un apposito blocco **`init`**

```
class Triangolo(val a:Double, val b:Double, val c:Double) {
   // prosegue codice del costruttore primario
   if (condizione di esistenza violata)
       throw new IllegalArgumentException(…);
 ...
}
```

Scala

```
class Triangolo(val a:Double, val b:Double, val c:Double) {
   // blocco init completa codice del costruttore primario
   init {
     if (condizione di esistenza violata)
       throw IllegalArgumentException(…);
   }
  ...
}
```

Kotlin

Definizione di nuovi tipi di eccezione (e loro lancio deliberato)

DEFINIRE NUOVI TIPI DI ECCEZIONI

- Negli esempi precedenti sui costruttori, l'eccezione lanciata è stata *scelta fra quelle predefinite*
  - **`IllegalArgumentException`** è un classico per gli argomenti
  - MA proprio perché molto usata, non descrive con chiarezza la situazione specifica che si è verificata
- Per renderla più specifica, si può innanzitutto *personalizzare il messaggio incapsulato:*

```
throw new IllegalArgumentException(
                          "triangolo impossibile")
```

- Tuttavia, sarebbe ancora meglio *avere proprio un tipo di eccezione ad hoc, diverso da quelli standard*

ESEMPIO: TRIANGOLO IMPOSSIBILE

Java

```
public Triangolo(double a, double b, double c)
        throws IllegalArgumentException {

    if (a>=b+c || b>=a+c || c>=a+b) {
      throw new IllegalArgumentException(
        "lati assurdi: "+a+", "+b+", "+c);
    }
    // happy path: inizializza normalmente
    this.a=a; this.b=b; this.c=c;
}
```

Modifica analoga negli altri linguaggi

Il messaggio d'errore specifico:

- costruisce un oggetto-eccezione dettagliato e preciso
- permette al cliente una diagnostica migliore

Se nella stessa applicazione ci sono anche Frazioni e altri oggetti, è tutta una IAE!

MA il cliente è comunque inondato di **`IllegalArgumentException`**

DEFINIRE NUOVI TIPI DI ECCEZIONI

- È possibile definire *nuovi tipi di eccezione,* con nomi a nostra scelta, per rappresentare nuovi «tipi di errore»
  - il *nome* dell'eccezione descrive con chiarezza l'accaduto
  - si può scegliere se controllo obbligatorio o facoltativo
  - *il cliente può predisporre un catch ad hoc per questo tipo di problema, diverso da ogni altro*
- Ad esempio, per il triangolo impossibile si potrebbe inventare **`ImpossibleTriangleException`**
  - una specializzazione di **`IllegalArgumentException`**
  - controllo facoltativo, ma *nome più specifico e descrittivo*
  - eventualmente, anche con costruttori speciali o dati extra
- Analogamente, per **`Frazione`** si potrebbe inventare **`ZeroDenException`** o qualcosa di simile

ImpossibleTriangleException

Java

```
public class ImpossibleTriangleException
   extends IllegalArgumentException {

 public ImpossibleTriangleException() { super(); }

 public ImpossibleTriangleException(String s){ super(s); }

}
```

~C#

Scala

```
class ImpossibleTriangleException(message: String)
       extends Exception(message) {}
```

Kotlin

```
class ImpossibleTriangleException(message: String)
        : Exception(message)
```

- In Java, il controllo facoltativo richiede che **`ImpossibleTriangleException`** derivi da **`IllegalArgumentException`** (da **`Exception`** se obbligatorio)
- Negli altri linguaggi si deriva tipicamente da **`Exception`**
- Ciò che conta comunque è avere un *nuovo tipo di eccezione*, con *nome distinto*

ESEMPIO: TRIANGOLO IMPOSSIBILE

```
public Triangolo(double a, double b, double c)
       throws ImpossibleTriangleException {


   if (a>=b+c || b>=a+c || c>=a+b)
     throw new ImpossibleTriangleException(
       "lati assurdi: "+a+", "+b+", "+c);
   // happy path: inizializza normalmente
   this.a=a; this.b=b; this.c=c;
}
```

Java

~C#

Inattesa, quindi in Java per prassi si indica

- Ora **`Triangolo`** ha la sua eccezione specifica
  - il cliente potrà intercettarla con un *catch specifico*, che catturi *solo quella,* senza rischiare di confonderla con altre "illegal argument"
- Analogamente si procede per **`Frazione`**

ESEMPIO: TRIANGOLO IMPOSSIBILE

```
class Triangolo(val a:Double, val b:Double, val c:Double){
   if (a>b+c || b>a+c || c>a+b)
        throw new ImpossibleTriangleException(
         "lati assurdi = " + a + ", " + b + ", " + c);
...
```

Scala

```
public open class Triangolo(val a:Double, val b:Double,
                            val c:Double)  {
  init {
    if (a>b+c || b>a+c || c>a+b)
       throw ImpossibleTriangleException(
         "lati assurdi = " + a + ", " + b + ", " + c);
  }
...
```

Kotlin

UN NUOVO MAIN in Java e C#

Java

~C#

```
public static void main(String[] args){
List<Triangolo> triangoli = new ArrayList<>();
  double[][] terneDiLati = {{3,4,5},{1,4,1},{3,3,3}};
  for (double[] terna : terneDiLati) {
    try {
      Triangolo trg = new Triangolo(terna[0],terna[1],terna[2]);
      triangoli.add(trg);
      System.out.println(trg);
    }
    catch (ImpossibleTriangleException e) {
      // gestione eccezione: nulla da fare
      System.out.println(e);
    }
  }
  Sy
}
```

Impossibile

Controllo sempre facoltativo: il try/catch non è indispensabile, *ma permette di proseguire*

Politica mirata alla *riduzione del danno:* il triangolo impossibile non è incluso nella lista

```
Triangolo di area 6.0 e perimetro 12.0
ImpossibleTriangleException: lati assurdi = 1.0, 4.0, 1.0
Triangolo di area 3.897114317029974 e perimetro 9.0
```

UN NUOVO MAIN in Scala e Kotlin

Scala

```
def main(args: Array[String]) : Unit = {
  try {
       val t1 = new Triangolo(3,4,5); println(t1);
       val t2 = new Triangolo(1,4,1); println(t2);
  }
  catch {
    case e: ImpossibleTriangleException => {
```

Impossibile

Kotlin

```
fun main(args: Array<String>) {
  try {
       val t1 = Triangolo(3,4,5); println(t1);
       val t2 = Triangolo(1,4,1); println(t2);
    }
  catch(e: ImpossibleTriangleException){
       println(e);
       kotlin.system.exitProcess(2);
  }
}
```

Impossibile

```
Triangolo di area 6.0 e perimetro 12.0
ImpossibleTriangleException: lati assurdi = 1.0, 4.0, 1.0
```

GESTORI MULTIPLI

- *Distinguere* i tipi di allarmi è utile per *gestirli diversamente*
  - ad esempio, triangolo impossibile vs frazione con denominatore zero

Java

```
public static void main(String[] args){
   ...
   for (int i=0; i<dati.length; i++) {
       try {
         trg[i] = new Triangolo(…);
         System.out.println(i+ ") " + trg[i]);
       }
       catch (ImpossibleTriangleException e) {
         // grave: non c'è nulla che si possa fare
       }
       catch (ZeroDenominatorException e) {
         // gestione diversa: lo sostituiamo con infinito (o "molto grande")
       }
  }
}
```

Gestione differenziata nei due casi

RIASSUMENDO…

- Una eccezione *è un normalissimo oggetto,* istanza di
  - **`Exception`** o sottoclassi, nel caso di *controllo obbligatorio*
  - **`RuntimeException`** o sottoclassi, nel caso di *controllo facoltativo*
- Come ogni classe, prevede costruttori e metodi (e dati)
  - costruttore di default
  - costruttore con argomento stringa (il messaggio d'errore associato)
  - metodo **`getMessage`** per recuperare il messaggio d'errore
  - campo dati **`bytesTransferred`** in **`InterruptedIOException`**
- Possiamo facilmente *definire noi nuovi tipi di eccezioni* per rappresentare situazioni nuove, errori logici, etc.

Rilancio di eccezioni

RILANCIO DI ECCEZIONI

- È possibile che *non si sappia come gestire una eccezione,* perché nel metodo dove si verifica l'errore *non si hanno informazioni sufficienti per gestirla*
  - non ha senso un catch «tanto per metterci qualcosa, a buon senso»
  - meglio delegare la gestione chi possa farla *con cognizione di causa*
- Perciò *si può decidere di NON gestire una eccezione* anche a controllo obbligatorio, *"lasciandola fuoriuscire"*
  - nel metodo non ci sarà allora alcun `try/catch`
- MA in tal caso:
  - in Java, la signature deve contenere *un avvertimento* che quel metodo può "sparare fuori" un'eccezione → *dichiarazione* **`throws`**
  - è *il chiamante* a dover gestire l'eccezione con **`try/catch`** (o magari decidere di rilanciarla a sua volta..)

RILANCIO DI ECCEZIONI: `throws`

Ad esempio, un metodo che apra un file può decidere di *far gestire* ***`FileNotFoundException`*** *all'esterno,* ma allora deve *avvisare*:

Java

```
void read(String fname) throws FileNotFoundException{
    FileReader f = new FileReader(fname);
}
```

NB: *negli altri linguaggi il rilancio non richiede alcuna dichiarazione throws, che non esiste*

- Potendo lanciare un'eccezione a controllo obbligatorio, il costruttore di **`FileReader`** richiederebbe **`try/catch`**
- Optando per rilanciarne la gestione all'esterno, *deve avvisare i clienti del pericolo* → in Java, dichiarazione **`throws`** obbligatoria
- Chi chiamerà questa funzione **`read`** dovrà gestire l'eccezione
  - o invocando **`read`** in un **`try/catch`**
  - o rilanciandola all'esterno a sua volta

RILANCIO DI ECCEZIONI: `throws`

Ad esempio, un metodo che apra un file può decidere di *far gestire* ***`FileNotFoundException`*** *all'esterno,* ma allora deve *avvisare*:

Java

```
void read(String fname) throws FileNotFoundException{
    FileReader f = new FileReader(fname);
}
```

NB: *negli altri linguaggi il rilancio non richiede alcuna dichiarazione throws, che non esiste*

Java

```
public void printAll(String filename){
  try{
    read(filename); // è pericolosa!
  }
  catch(FileNotFoundException e){
    …
  }
}
```

Qui il cliente sceglie di catturare e gestire l'eccezione

RILANCIO DI ECCEZIONI: `throws`

Ad esempio, un metodo che apra un file può decidere di *far gestire* ***`FileNotFoundException`*** *all'esterno,* ma allora deve *avvisare*:

Java

```
void read(String fname) throws FileNotFoundException{
    FileReader f = new FileReader(fname);
}
```

NB: *negli altri linguaggi il rilancio non richiede alcuna dichiarazione throws, che non esiste*

Java

```
public void printAll(String filename)
                    throws FileNotFoundException {
    read(filename); // è pericolosa!
    ...
}
```

NB: *negli altri linguaggi il rilancio non richiede alcuna dichiarazione throws, che non esiste*

Qui invece il cliente sceglie di rilanciarla all'esterno a sua volta

Sarà allora il chiamante di costui a doversene far carico

Incapsulamento di eccezioni

ECCEZIONI FISICHE vs ECCEZIONI LOGICHE

- Una *eccezione «fisica»* esprime l'accaduto, ma *può non farlo col livello di astrazione richiesto*
- Per questo si definiscono *eccezioni «logiche»*
  1. La *stessa eccezione «fisica»* può corrispondere a *situazioni «logiche» diverse*
  2. Dualmente, a *più eccezioni «fisiche» distinte* può corrispondere *la stessa situazione «logica»*
  3. Può anche solo essere *conveniente* «convertire» *un'eccezione «fisica» generica* in una *eccezione logica più specifica*, che meglio descriva l'accaduto.

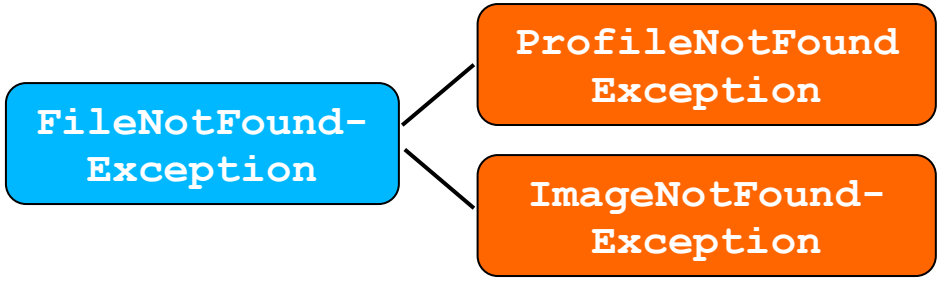
SCENARI POSSIBILI

1. Unica *eccezione «fisica»* → *eccezioni «logiche» diverse*
   - qualunque file mancante causa una **`FileNotFoundException`**
   - ma per noi la mancanza *di un file piuttosto che di un altro* potrebbe indicare *situazioni logiche molto diverse*
2. *Eccezioni «fisiche» distinte* → *unica eccezione «logica»*
   - la mancata apertura di un file o la sua errata struttura interna sono *problemi «fisici» diversi*
   - ma per noi potrebbero essere *due forme diverse* dello *stesso problema logico – l'impossibilità di accedere ai dati*
3. Conversione di *un'eccezione «fisica» generica* in una *eccezione logica più specifica* e significativa
   - **`FileNotFoundException`** indica il fatto fisico, ma se stiamo cercando di caricare *un'immagine* può essere utile propagare un'eccezione *logica «nostra»*, come **`ImageNotFoundException`**

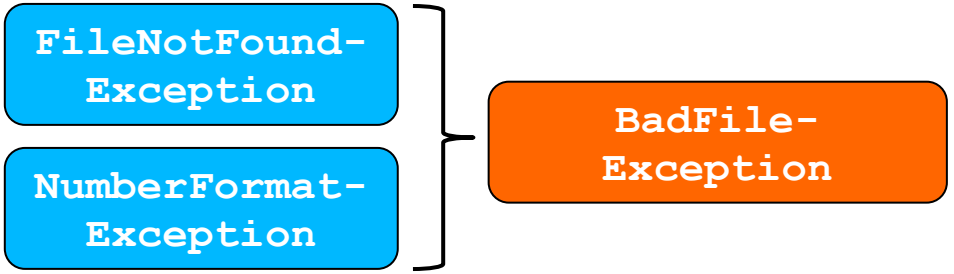
INCAPSULAMENTO DI ECCEZIONI

Per questi motivi, spesso si *cattura un'eccezione «fisica»* per *rilanciarne subito una «logica» diversa*

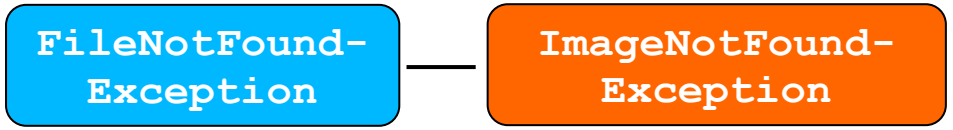
1. separare un'eccezione fisica in più eccezioni logiche distinte



2. accorpare più eccezioni fisiche in un'unica eccezione logica



3. convertire un'eccezione fisica «generica» in una eccezione logica *più specifica*



ESEMPIO 1

Java

```
public void readUserData(String user)
  throws ProfileNotFoundException, ImageNotFoundException{
  try {
   FileReader f1 = new FileReader(user+".txt");
      ...
  } catch(FileNotFoundException e1){
     throw new ProfileNotFoundException(e1);
  }
  try {
   FileReader f2 = new FileReader(user+".png");
     ...
  } catch(FileNotFoundException e2){
     throw new ImageNotFoundException(e2);
  }
}
```

Due eccezioni *logiche* distinte

Eccezione *fisica*

Qui lancia un primo tipo di eccezione *logica..*

Eccezione *fisica*

.. e qui un tipo di eccezione *logica diverso*

ESEMPIO 2

Java

```
public void readUserData(String filename)
            throws BadFileFormatException {
  try {
   FileReader f = new FileReader(filename);
   String s = null;
      ... // lettura linea di testo
   int n = Integer.parseInt(s);
  } catch(FileNotFoundException e1){
     throw new BadFileFormatException(e1);
  }
  } catch(NumberFormatException e2){
     throw new BadFileFormatException(e2);
  }
}
```

Eccezione *logica* unica

Eccezione *fisica* di un tipo

Eccezione *fisica* di un altro tipo

Accorpate in un *unico tipo* di eccezione *logica*

ESEMPIO 2 – VARIANTE

Questa situazione è molto frequente, quindi da Java 7 è prevista una *sintassi scorciatoia (shortcut):*

Java

```
public void readUserData(String filename)
            throws BadFileFormatException {
  try {
   FileReader f = new FileReader(filename);
   String s = null;
      ... // lettura linea di tes
   int n = Integer.parseInt(s);
  }
  catch(FileNotFoundException | NumberFormatException e){
     throw new BadFileFormatException(e);
  }
}
```

SHORTCUT: *unico catch* per più eccezioni

MULTI-CATCH nei vari linguaggi

Escluso Kotlin, che al momento non offre esplicito supporto, il *multi-catch* è offerto praticamente in tutti i linguaggi:

Java

```
catch (Exception1 || Exception2 ||… e) {
  ...
}
```

C#

```
catch (Exception e)
   when (e is Exception1 || e is Exception2 || …){
     ...
}
```

NB: *costrutto valido a partire da C# 6.0*

Scala

```
catch {
   case e @ (_ : Exception1 | _ : Exception2 | …)
        => {...}
}
```

ESEMPIO 3

- In questo caso, fisicamente, è un classico “file non trovato”
- Però, trattandosi un’immagine, decidiamo di incapsularla in un *nuovo tipo di eccezione logica* che descriva meglio l‘accaduto
- Così, *il cliente potrà gestire ad hoc questa specifica situazione*

Java

```
public Image readImage(String filename)
                    throws ImageNotFoundException {
  try {
   FileReader f = new FileReader(filename);
      ...
  } catch(FileNotFoundException e){
     throw new ImageNotFoundException(e);
  }
}
```

Eccezione “fisica”

Eccezione “logica”

ESEMPIO 4

- Riprendendo l'esempio del parsing di numeri e valute..
- Il metodo **`parse`** a due argomenti non lancia eccezioni
  - però, il secondo argomento, **`ParsePosition`**, dice "fin dove è arrivato" il parsing → *basta controllare se è in fondo alla stringa*
  - ergo, se non è arrivato in fondo, la stringa è sbagliata *anche se il metodo ha fatto passare la cosa "sotto silenzio"*
- Meglio intercettare la cosa e *lanciare noi la giusta eccezione*

Java

```
ParsePosition position = new ParsePosition(0);
Number n = f.parse("10-10", position);
if (position.getIndex() != str.length()) {
 throw new OpportunaEccezione(…);
}
```

Se il parsing non arriva in fondo, provvediamo noi a lanciarela "giusta" eccezione

Leggendo da file, probabilmente una **`BadFileFormatException`**

RIASSUMENDO..

- L'eccezione rappresenta un *allarme*
  - interrompe il normale flusso di controllo
  - causa l'entrata in scena di un *gestore di eccezione prestabilito*
- Alcuni linguaggi, come Java, distinguono fra controllo *obbligatorio* o *facoltativo*
  - a controllo obbligatorio se rappresenta una *situazione esterna, al di fuori del nostro controllo* (non un bug)
  - a controllo facoltativo se rappresenta invece una *violazione di precondizione* (contratto violato), ossia un bug
- Può essere di tipo *standard* o *definito da noi*
  - il tipo definito da noi aiuta a *differenziare l'allarme da altri* permettendone una cattura e gestione specifiche

TECNICAMENTE…

- Un'eccezione è un normale oggetto
  - in Java, istanza di *una qualche sottoclasse* di **`Exception`**
  - se deriva da `RuntimeException` → controllo *facoltativo*
  - se non deriva da `RuntimeException` → controllo *obbligatorio*
- Un metodo che *possa lanciare* un'eccezione
  - in Java, deve dichiararla nella signature con la clausola **`throws`** SOLO se è a controllo *obbligatorio (altrimenti, può non farlo)*
  - negli altri linguaggi, non esistono dichiarazioni nella signature
- È possibile lanciare un'eccezione *deliberatamente*
  - COME: creando prima l'opportuno oggetto-eccezione con **`new`** e poi lanciandolo con l'istruzione **`throw`**
  - PERCHÉ: per segnalare un «errore logico» o per incapsulare un'eccezione «fisica» generica in una «logica» più specifica