# Eccezioni

Dal materiale di Stefano Ferretti
s.ferretti@unibo.it

# Eccezioni

- Quando un programma viene **eseguito** possono verificarsi **errori** o **anomalie**
- Esempi:
  - Si cerca di accedere tramite il metodo **`get`** a un elemento di un **`ArrayList`** fuori dai limiti
    - Stessa cosa per gli array, ad es. A[**-1**] o A[**A.length**]
  - Si cerca di interpretare una stringa numerica con il metodo **`parseInt`**, ma la stringa non contiene la rappresentazione di un numero intero
  - …

# Eccezioni

- Un programma scritto bene dovrebbe **gestire** il più possibile queste situazioni
- E' un compito **difficile** in generale
- Può essere che al punto del programma dove si verifica l'errore non si abbiano le info. per **gestirlo** e ripristinare uno **stato corretto**

# Eccezioni

- Ad es. i metodi **`get`** di **`ArrayList`** e **`parseInt`** non hanno sufficiente informazione per sapere come gestire un errore durante l'**esecuzione** del programma (*runtime*)
  - Si dovrebbe chiedere all'utente di fare un'altra operazione?
  - Si dovrebbe terminare il programma?
- Queste decisioni sono **indipendenti** da quello che deve fare il metodo da cui sono **originati** gli errori
  - Non è *responsabilità* del metodo
- È da *qualche altra parte* nel programma, scorrendo la pila delle attivazioni, che si può arrivare ad un punto in cui si può **gestire** la situazione anomala

# Eccezioni

- Il meccanismo delle **eccezioni** fornisce primitive per gestire in modo flessibile situazioni anomale

- Il **controllo** dell'esecuzione viene passato dal punto in cui si **verifica** l'errore a "*un altro punto del programma*" dove l'errore può essere gestito nel modo più opportuno
  - Più dettagli in seguito

# Valore di ritorno

- Un approccio alternativo è **segnalare** eventuali anomalie ritornando un'apposito **valore di ritorno**
  - Ad es. se cerco di creare un oggetto di una classe ma per qualche motivo non ci riesco, il metodo che esegue questa operazione può restituire il valore speciale **`null`**
  - Per gestire l'eventuale errore il codice che chiama questo metodo deve preoccuparsi di **controllare** il valore di ritorno e, nel caso sia **`null`**, fare qualcosa per gestire la situazione

# Valore di ritorno

- Questo approccio, usato ad es. nei programmi scritti in **C**, presenta alcuni inconvenienti
  - Il programmatore deve **codificare** sia il comportamento normale che **tutti** i possibili errori
  - Il codice risulta molto **complicato** e **poco leggibile**
  - Il valore di ritorno va controllato **ogni volta** che il metodo è richiamato
  - La segnalazione di errore **potrebbe non bastare**: il chiamante potrebbe **non** avere tutte le informazioni per gestire l'errore e sarebbe costretto a risegnalare esplicitamente l'errore al suo chiamante e così via…

# Eccezioni

- Usare il valore di ritorno per gestire gli errori è accettabile in certi contesti, ma in generale è bene utilizzare le **eccezioni**

- Questo meccanismo è progettato per risolvere due problemi principali:
  1. Le eccezioni **non** possono essere **trascurate**
  2. Le eccezioni devono essere **gestite** da un "gestore" **competente**, non necessariamente dal *chiamante* del metodo che fallisce

# Sollevare un'eccezione

- Quando si verifica una situazione inaspettata bisogna "**sollevare**" o "**lanciare**" un'eccezione

- Un'eccezione è un **oggetto** di una classe
  - Per sollevare un'eccezione si usa il costrutto **`throw`** seguito dall'oggetto di tale classe
    - Un metodo *throwable* non è necessariamente un'eccezione

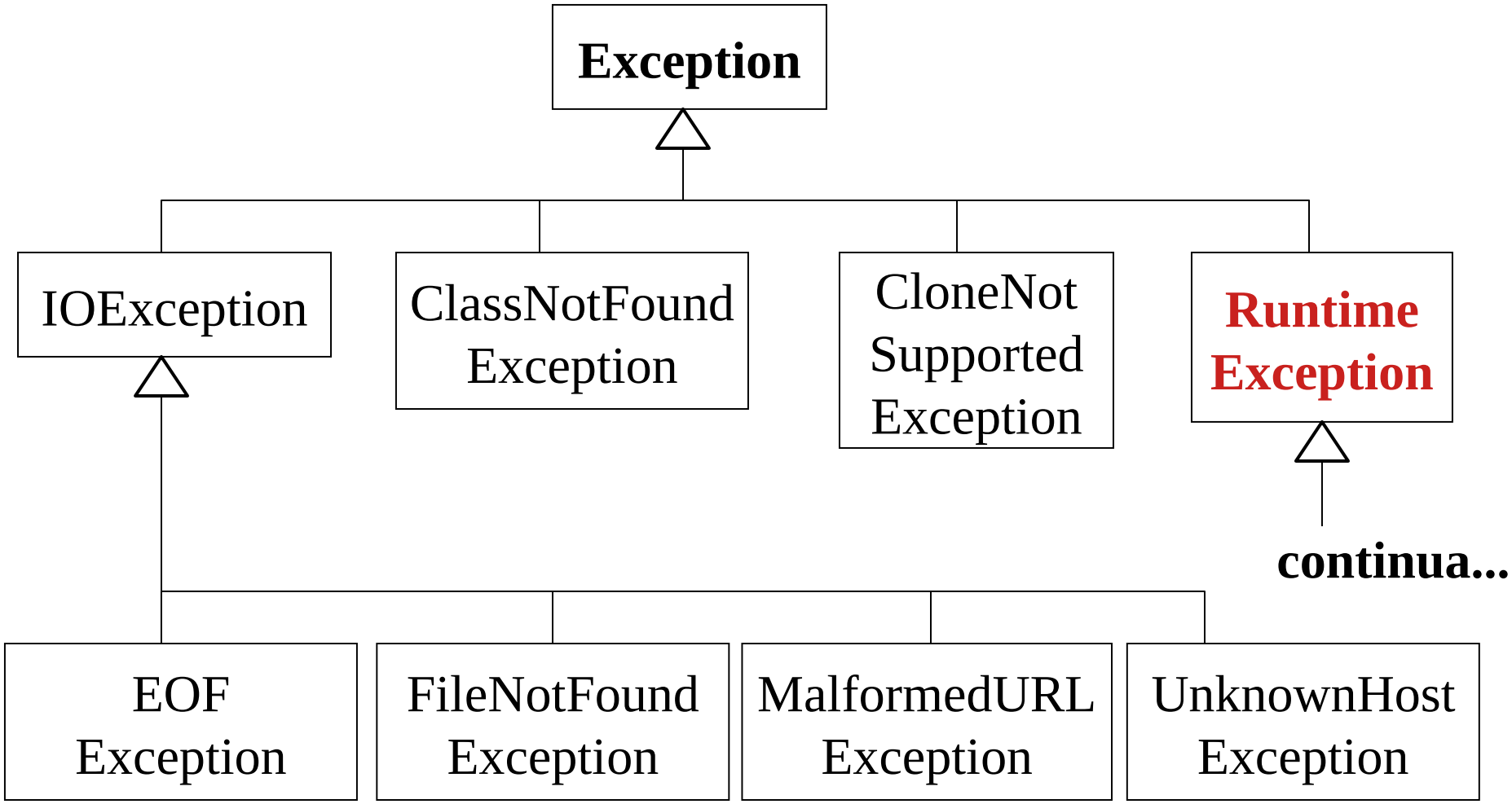
  - Esempio: un utente chiede un prelievo maggiore del saldo del suo conto attraverso un metodo un certo metodo **`withdraw(double amount)`**

# Sollevare un'eccezione

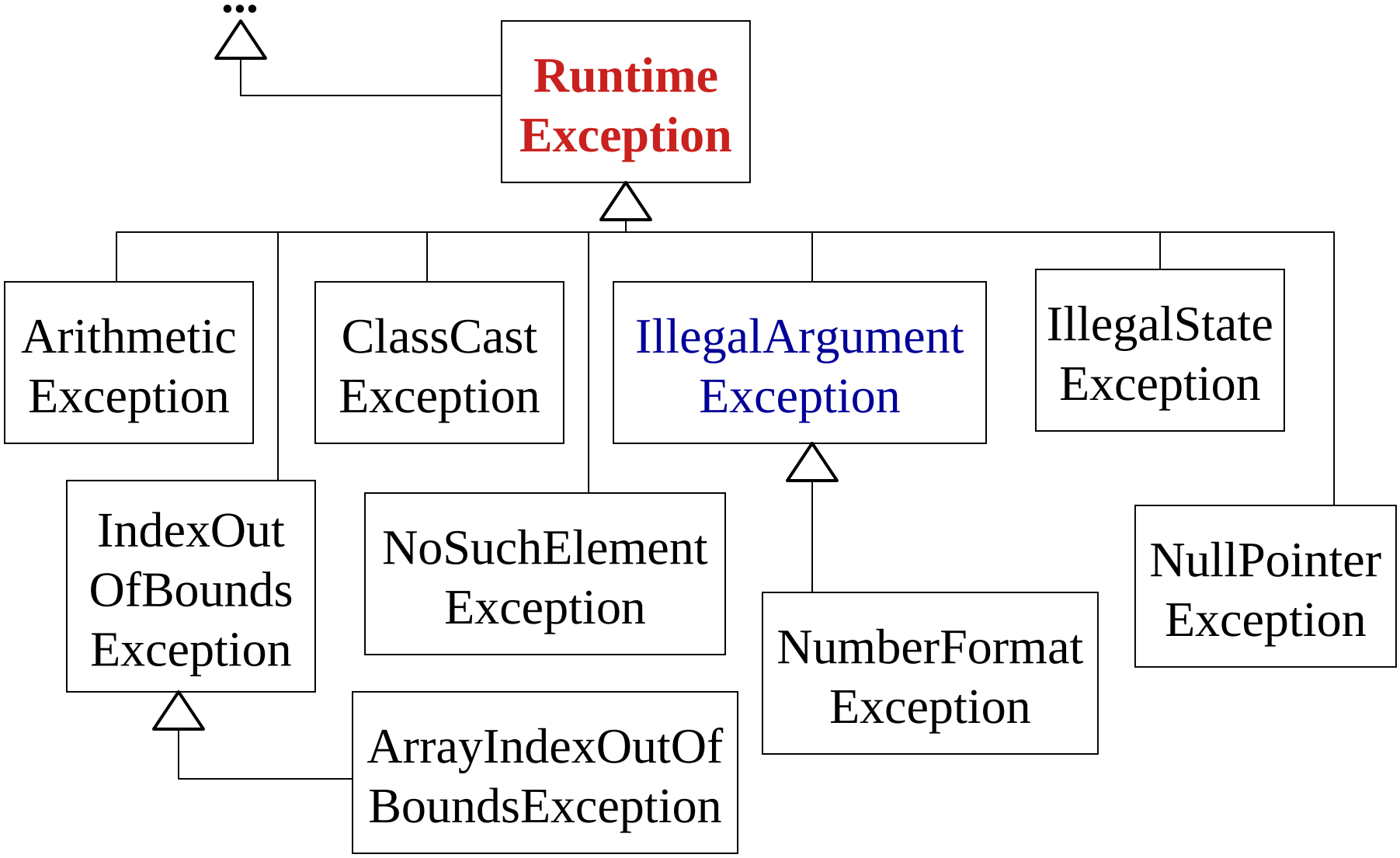
- La libreria standard di Java ha una **gerarchia di classi** per gestire eccezioni di diversa natura
  - Scegliamo quella che fa più al caso nostro, creiamo l'oggetto eccezione e lo "lanciamo"

- Nell'es. precedente, l'eccezione più adatta a descrivere la richiesta di prelievo maggiore del saldo è **`IllegalArgumentException`** in quanto è l'argomento **`amount`** del metodo **`withdraw`** a causare il problema

# Gerarchia di eccezioni (incompleta)



# Gerarchia di eccezioni (incompleta)



# Esempio

```
public class BankAccount {
 public void withdraw(double amount) {
  if (balance < amount) {
    // Parametro amount troppo grande
    IllegalArgumentException exception =
            new IllegalArgumentException(
              “Amount exceedes balance”);
    throw exception;
  } else ...
```

# Oppure

```
public class BankAccount {
 public void withdraw(double amount) {
  if (balance < amount) {
    // Parametro amount troppo grande
    throw new IllegalArgumentException(
              “Amount exceedes balance”);
  } else ...
```

# Semantica del throw

- Quando viene eseguita l'istruzione **`throw`** il metodo si **arresta** immediatamente
- Il controllo **non** torna di default al chiamante, ma viene passato al **gestore** dell'eccezione (*exception handler*) corrispondente
  - Procedimento **diverso** da invocazione metodo
- Per ora lasciamo in sospeso *come* questo gestore viene individuato

# Quando sollevare eccezioni?

- Consideriamo ad es. il metodo **`readLine`** della classe **`BufferedReader`**
  - **`https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/io/BufferedReader.html#readLine()`**
- Se si incontra la fine del file ritorna **`null`**
- Perché non lancia una **`EOFException`**?
  - Il motivo è che il fatto che un file termini **non** è un evento eccezionale! *Ogni* file termina…

# Quando sollevare eccezioni?

- L'eccezione **`EOFException`** va lanciata **solo** se la fine del file giunge **inaspettatamente**
- Ad es. l'EOF viene raggiunto ma la sequenza di dati in input è **incompleta**
- Qui l'errore è dovuto a un caso **eccezionale**, ad es. il file che si sta leggendo è corrotto

# Quando sollevare eccezioni?

- In generale si deve **valutare** se l'evento che si vuole segnalare è un caso **eccezionale** oppure una condizione "*normale*" che si verifica regolarmente
- Solo nel **1o caso** è opportuno lanciare un'eccezione
- Nel **2o caso** meglio usare un valore di ritorno che codifichi l'evento

# Rilanciare eccezioni

- Il metodo **`readLine`** può, ad es., anche lanciare una **`IOException`**
  - Queste eccezioni sono **controllate** (*checked*): ogni metodo che lancia un'eccezione checked *deve* essere o **catturata** o **dichiarata** nel metodo dove è sollevata
  - Le eccezioni **non controllate** (*unchecked)* non hanno questo requisito

- Se in un metodo chiamiamo **`readLine`** su un oggetto della classe **`BufferedReader`** può accadere che **`readLine`** sollevi l'eccezione

# Eccezioni (non) controllate

- Le eccezioni **controllate** sono pensate per modellare situazioni dove il programmatore **non ha responsabilità**
  - Es. la lettura di un file può interrompersi per cause esterne (un guasto a un hard disk, un collegamento di rete che si interrompe, etc.)

- Le eccezioni **non controllate** invece rappresentano un **errore** del programmatore
  - "Se è una eccezione **`RuntimeException`** (*o figlia*) è colpa tua!"
  - Es. la **`NullPointerException`** *non* dipende da cause esterne: un buon codice dovrebbe controllare se un riferimento è nullo prima di utilizzarlo

# Rilanciare eccezioni

- Per le eccezioni checked il **compilatore impone** che il metodo dichiari cosa fare
- In generale abbiamo due scelte:
  1. Definire un **gestore di eccezioni** per gestirle
  2. Decidere che il metodo non è “competente”: diciamo al compilatore che in caso di eccezione il metodo deve **terminare** immediatamente e l’eccezione deve essere “**rilanciata**” (o *propagata*) al metodo **chiamante**
     - Il metodo chiamante potrà a sua volta gestirla o rilanciare quel tipo di eccezione e così via…

# Rilanciare eccezioni

- Per segnalare al compilatore che il nostro metodo **non** gestisce eccezioni **checked** si usa la keyword **`throws`**:

```
public class Coin {
  public void read(BufferedReader in)
  throws IOException {
    // Se IOException, rilanciata
    double value =
        Double.parseDouble(in.readLine());
    name = in.readLine();
  }
  ...
}
```

# Esempio

```
public class Purse {
  public void read(BufferedReader in)
                        throws IOException {
    while (...) {
      Coin c = new Coin();
      // read di Coin rilancia!
      c.read(in);
      add(c);
    }
...
}
```

# Rilanciare eccezioni

- All'interno di un metodo possono verificarsi **più** eccezioni checked

- La clausola **`throws`** può essere usata con una lista di eccezioni separate da virgola:

```
public void m() throws IOException,
                       ClassNotFoundException,
                                              ... {
...
```

# Rilanciare eccezioni

- Un metodo che non gestisce una o più eccezioni può sembrare “irresponsabile”
  - “*scaricabarile*”
- Tuttavia è **giusto** che sia così **se** il metodo non può gestire correttamente l’errore!
  - Pensiamo a un metodo **`read`** di basso livello come quelli visti come esempi
  - Come dovrebbe gestire l’**`IOException`**?

# Rilanciare eccezioni

- Dovrebbe stampare un messaggio su std output?
- E se la nostra classe opera in un sistema dove l'utente non vede affatto lo standard output?
  - es. distributore automatico
- E se l'utente non comprendesse il messaggio?
- *Morale*: se non si è sicuri di quello che si fa, meglio delegare ad un metodo **competente**

# Gestori di eccezioni

- **Tutte** le eccezioni che si possono verificare in una certa applicazione *dovrebbero* essere gestite da qualche parte

- Se un'eccezione non viene catturata da alcun gestore il programma **termina** stampando su std error la **pila di attivazioni** che l'eccezione ha attraversato prima della terminazione

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at A.f(A.java:6)
        at A.main(A.java:10)
```

# Gestori di eccezioni

- Un gestore di eccezioni è definito da un **blocco `try-catch`**
- Se un comando all'interno di un **blocco `try`** lancia un'eccezione allora il **tipo** dell'eccezione viene **confrontato** con i tipi elencati nelle **clausole `catch` associate** al blocco **`try`**
- Se uno dei tipi nelle clausole **`catch`** è **compatibile** (*uguale o superclasse*) col tipo dell'eccezione allora la **propagazione** dell'eccezione si **arresta**: viene eseguito il corrispondente blocco **`catch`** e l'esecuzione continua a partire **dalla fine** del blocco **`try-catch`**
  - L'esecuzione si arresta al **primo** catch compatibile, se esiste

# Gestori di eccezioni

```
try {
  BufferedReader in = new BufferedReader(new
          InputStreamReader(System.in));
  System.out.println("How old are you?");
  String inputLine = in.readLine();
  int age = Integer.parseInt(inputLine);
  age++;
  System.out.println("Next year you'll be "  + age);
}
catch (IOException exception) {
  System.out.println("I/O exception " + exception);
}
catch (NumberFormatException exception) {
  System.out.println("Input was not a number");
}
```

**readLine**

```
public String readLine()
                throws IOException
```

Reads a line of text. A line is considered to be terminated by any one of a line feed ('\n'), a carriage return ('\r'), a carriage return followed immediately by a line feed, or by reaching the end-of-file (EOF).

**Returns:**

A String containing the contents of the line, not including any line-termination characters, or null if the end of the stream has been reached without reading any characters

**Throws:**

`IOException` - If an I/O error occurs

- Il metodo **`readLine()`** rilancia una **`IOException`**
  - Nella sua implementazione ci sarà qualcosa tipo **`if (…) throw new IOException(…);`**
- Non sapendo gestire l'eccezione adeguatamente, la **rilancia**
- **Deve** essere gestita dai metodi che usano **`readLine()`**:
  - Dichiarando **`throws IOException`**, oppure
  - Con un blocco **`try-catch`** che catturi le **`IOException`**

# Gestori di eccezioni

- Nell'es. precedente catturiamo eccezioni di tipo:
  - **`IOException`**
    - può essere lanciata da **`readLine`**
  - **`NumberFormatException`**
    - può essere lanciata da **`parseInt`**
  - Eventuali loro **sottotipi**
- La gestione, semplice in questo caso, consiste solo nell'informare via std output l'utente dell'accaduto
  - Si potrebbe ad esempio migliorare la gestione di **`NumberFormatException`** permettendo all'utente di riprovare a inserire il numero

# Gestori di eccezioni

- La clausola **`catch(`***Type* **`ex) {`** *Blocco istruzioni* **`}`**
  viene attivata **solo** quando all'interno del blocco **`try`** corrispondente viene lancia un'eccezione di tipo *Type* (o di un **sottotipo** di *Type*)
- L'**attivazione** della clausola **`catch`** assegna alla variabile **`ex`** il **riferimento** all'oggetto eccezione che è stato lanciato ed esegue il blocco di istruzioni
- Dopo aver eseguito le istruzioni del blocco l'esecuzione prosegue normalmente **dopo** il blocco **`try`**
  - **Tutte** le clausole **`catch`** **successive** a quella attivata vengono **ignorate**!

# Consigli

- Evitare di definire gestori di eccezioni troppo **generici**
  - **`catch(Exception e)`** o addirittura
  - **`catch(Throwable t)`**
- Mai zittire il compilatore, che lamenta la mancata gestione di un'eccezione controllata, inserendo un gestore di eccezioni che non fa **nulla**
- Evitare che la propria applicazione possa lanciare eccezioni di tipo **`RuntimeException`**
  - E sottotipi di **`RuntimeException`**

# La clausola `finally`

- A volte è utile specificare operazioni da compiere *sia* nel caso di esecuzione normale *che* in caso di eccezione: questo è gestito dalla clausola **`finally`**

- **`finally`** è abbinata ad un blocco **`try-catch`** e, in **qualunque** modo il controllo lasci il blocco…
  - return, eccezione non gestita, eccezione catturata, terminazione normale
- …i comandi del blocco **`finally`** vengono **sempre eseguiti** immediatamente **prima** di abbandonare il blocco

# La clausola **finally**

```
try {
   …
}
catch (e1 MyException1) { … }
…
catch (eN MyExceptionN) { … }
finally {
 … // Questo blocco viene
   // SEMPRE eseguito!
}
```

# La clausola `finally`

- **`finally`** è molto utile per effettuare operazioni di "controllo e pulizia" come ad es. la chiusura di file:

```
BufferedReader in;
try {
 in = new BufferedReader(newFileReader(fileName));
 purse.read(in);
}
finally {
 if (in != null) in.close();
}
```

- Indipendentemente che la lettura vada a buon fine o meno, il file verrà sempre e comunque chiuso

# Esempio: calcolo dei consumi

```
1  public class Consumi {
2
3      public static void main(String[] args) {
4          java.util.Scanner tastiera = new java.util.Scanner(System.in);
5          int kilometri, litri, distanza;
6
7          System.out.print("Inserire i kilometri percorsi: ");
8          kilometri = tastiera.nextInt();
9
10         System.out.print("Inserire i litri di benzina consumati: ");
11         litri = tastiera.nextInt();
12
13         distanza = kilometri / litri;
14
15         System.out.println("La tua auto fa " + distanza + " kilometri al litro");
16
17         System.out.println("... fine del programma.");
18     }
19 }
```

# Esempio: calcolo dei consumi

Il programma potrebbe fare una divisione per zero!

## Esempio di esecuzione del programma:

```
Inserire i kilometri percorsi: 8
Inserire i litri di benzina consumati: 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Consumi.main(Consumi.java:13)
```

Per evitare il problema abbiamo a disposizione due approcci:

1. usare un blocco `if-else`
2. usare un blocco `try-catch`

# Esempio: calcolo dei consumi

Quando l'utente immette il valore per `litri`, il programma si accerta che tale valore non sia nullo (o negativo). In tal caso, non viene eseguita la divisione.

```
litri = tastiera.nextInt();

if (litri <= 0) {
    ...
}
else {
    distanza = kilometri / litri;
    ...
}
```

# Esempio: calcolo dei consumi

```
1  public class Consumi {
2
3      public static void main(String[] args) {
4          java.util.Scanner tastiera = new java.util.Scanner(System.in);
5          int kilometri, litri, distanza;
6
7          System.out.print("Inserire i kilometri percorsi: ");
8          kilometri = tastiera.nextInt();
9
10         System.out.print("Inserire i litri di benzina consumati: ");
11         litri = tastiera.nextInt();
12
13         if (litri <= 0) {
14             System.out.println("Impossibile: la tua auto non consuma benzina?!?");
15             System.out.println("Controlla meglio...");
16         } else {
17             distanza = kilometri / litri;
18             System.out.println("La tua auto fa " + distanza + " kilometri al
                   litro");
19         }
20
21         System.out.println("... fine del programma.");
22     }
23 }
```

# Esempio: calcolo dei consumi

La divisione per zero non viene mai effettuata!

## Esempio di esecuzione del programma:

```
Inserire i kilometri percorsi: 8
Inserire i litri di benzina consumati: 0
Impossibile: la tua auto non consuma benzina?!?
Controlla meglio...
... fine del programma.
```

# Esempio: calcolo dei consumi

Una divisione per zero produce un'eccezione. Invece che evitare l'esecuzione della divisione, la si può effettuare e si reagisce opportunamente all'eventuale eccezione.

```
1   try {
2       ...
3
4       distanza = kilometri / litri;
5
6       System.out.println("La tua auto fa " + distanza + " kilometri al litro");
7
8   } catch(Exception e) {
9       System.out.println("c'e' stato un problema: " + e.getMessage());
10      System.out.println("Possibile che la tua auto non consumi?!?");
11  }
```

# Esempio: calcolo dei consumi

La divisione per zero può succedere, ma viene gestita!

## Esempi di esecuzione del programma:

```
Inserire i kilometri percorsi: 8
Inserire i litri di benzina consumati: 2
La tua auto fa 4 kilometri al litro
... fine del programma.

Inserire i kilometri percorsi: 8
Inserire i litri di benzina consumati: 0
Forse hai un problema: / by zero
Possibile che la tua auto non consumi?!?
... fine del programma.
```

```java
public class Consumi {

  public static void main(String[] args) {
    java.util.Scanner tastiera = new java.util.Scanner(System.in);
    int kilometri, litri, distanza;

    try {
      System.out.print("Inserire i kilometri percorsi: ");
      kilometri = tastiera.nextInt();
      System.out.print("Inserire i litri di benzina consumati: ");
      litri = tastiera.nextInt();

      distanza = kilometri / litri;

      System.out.println("La tua auto fa " + distanza + " kilometri al litro");

    } catch (Exception e) {
      System.out.println("Forse hai un problema: " + e.getMessage());
      System.out.println("Possibile che la tua auto non consumi?!?");
    }

    System.out.println("... fine del programma.");
  }
}
```

# ArithmeticException

```
1  public class Consumi {
2
3    public static void main(String[] args) {
4      java.util.Scanner tastiera = new java.util.Scanner(System.in);
5      int kilometri, litri, distanza;
6
7      try {
8        System.out.print("Inserire i kilometri percorsi: ");
9        kilometri = tastiera.nextInt();
10       System.out.print("Inserire i litri di benzina consumati: ");
11       litri = tastiera.nextInt();
12
13       distanza = kilometri / litri;
14
15       System.out.println("La tua auto fa " + distanza + " kilometri al litro");
16     } catch (ArithmeticException e) { // selezioniamo l'eccezione
17       System.out.println("Forse hai un problema: " + e.getMessage());
18       System.out.println("Possibile che la tua auto non consumi?!?");
19     }
20     System.out.println("... fine del programma.");
21   }
22 }
```

Viene catturata solo l'eccezione specifica, le altre vengono rilanciate.

```
Inserire i kilometri percorsi: 30
Inserire i litri di benzina consumati: 3
La tua auto fa 10 kilometri al litro
... fine del programma.

Inserire i kilometri percorsi: 30
Inserire i litri di benzina consumati: 0
Forse hai un problema: / by zero
Possibile che la tua auto non consumi?!?
... fine del programma.

Inserire i kilometri percorsi: 30
Inserire i litri di benzina consumati: 3.0
Exception in thread "main" java.util.InputMismatchException
    at java.util.Scanner.throwFor(Scanner.java:909)
    at java.util.Scanner.next(Scanner.java:1530)
    at java.util.Scanner.nextInt(Scanner.java:2160)
    at java.util.Scanner.nextInt(Scanner.java:2119)
    at Consumi.main(Consumi.java:11)
```

# Esercizio

- Implementare l'esempio del calcolo dei consumi, catturando **tutte** le possibili **RuntimeException** che possono occorrere quando si usa `nextInt`
  - Per ognuna di queste stampare un corrispondente messaggio di errore
    - Vedere la documentazione di **`nextInt:`**

      https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/Scanner.html#nextInt()
  - Se si è verificata un'eccezione, l'inserimento va ripetuto

# Riferimenti

- Lucidi del Libro di Riferimento
- http://java.sun.com/
- http://www.dimi.uniud.it/mizzaro/dida/Prog0405

# Note sulla Licenza

- Parte di questi lucidi sono ispirati da lucidi sotto licenza Creative Commons
  - http://www.creativecommons.org/
  - I lucidi in questione sono tratti da http://www.dimi.uniud.it/~mizzaro/dida/Prog0405/
    - Per il corso di Programmazione e Laboratorio tenuto dal Prof. Stefano Mizzaro
- Di conseguenza, anche questi lucidi vengono distribuiti sotto licenza Creative Commons