# Java Exceptions

Nell'esecuzione dei programmi spesso si dà per scontato che il tutto venga eseguito senza errori ma può capitare che sorga la necessità di segnalare un malfunzionamento. Generalmente in tutti i linguaggi di programmazione un errore di questo tipo viene segnalato con un **messaggio di errore** spesso fraintendibile con una **segnalazione che spetta al chiamante e non al chiamato**.

Fortunatamente i linguaggi di programmazione moderni dedicano risorse per implementare costrutti in grado di gestire questi messaggi che vengono chiamati **eccezioni**. Una gestione efficiente delle eccezioni presenta quindi i seguenti vantaggi:

- separare la gestione degli errori dal codice applicativo
- consentire una propagazione controllata degli errori
- raggruppare o differenziare gli errori

Nel linguaggio di Java le eccezioni vengono gestite con il costrutto **try-catch**. In particolare nella **parte di try** si "prova" ad eseguire un segmento di codice, nella **parte di catch** si verifica se nel segmento di try è presente un particolare errore passato come parametro (errore prestabilito da espressioni costanti) agendo quindi in un certo modo se ciò si verifica.

```
try {
  FileInputStream fis = new FileInputStream(f);
  int b = fis.read();
  System.out.println(b);
} catch (FileNotFoundException e) {
    System.out.println("Non trovo il file...");
} catch (IOException e) {
    System.out.println("Errore di I/O");
}
```

# Eccezioni da programma

Un'eccezione è rappresentata da un oggetto di una specifica classe (sottoclasse di Exception), oggetto che contiene diverse informazioni circa l'errore che si sta verificando. In Java è possibile **sollevare esplicitamente** un'eccezione utilizzando il costrutto **throw**. Nell'esempio (immagine) precedente le eccezioni erano sollevate direttamente dal runtime system di Java.

Se un metodo solleva eccezioni queste devono **obbligatoriamente** essere **dichiarate**, questo procedimento viene effettuato direttamente dall'interfaccia del metodo.

```
public Object pop() throws EmptyStackException {
  if (!isEmpty()) return stack[--ptr];
  else throw new EmptyStackException();
}
```

# Procedura in risposta all'eccezione

Quando un'eccezione viene sollevata, indipendentemente dal Java runtime system o direttamente dall'utente con **throw**, viene immediatamente bloccata l'esecuzione del blocco di codice che ha causato l'eccezione. successivamente viene ripercorsa la catena di chiamate verificando volta per volta se esiste già un blocca try-catch che può risolvere l'eccezione. In caso affermativo viene eseguito il blocco in catch e l'esecuzione termina con le istruzioni nel blocco di catch. altrimenti la propagazione continua al chiamante del chiamante.

```
public class Test {
  public static void main(String[] args) {
    try {
      m1();
    } catch (TestException e) { … }
  }
  void m1() throws TestException { m2(); }
  void m2() throws TestException { m3(); }
  void m3() throws TestException {
    …
    throw new TestException();
  }
}
```

Come si può notare viene ricercato a catena l'exception sollevata nel metodo m3 che verrà risolta dal codice nella sezione "catch".

Java richiede che ogni metodo debba: o **gestire le eccezioni** che possono essere sollevate all'interno del proprio corpo **definendo un corpo formato da try e uno o più catch**, oppure **propagare tali eccezioni** dichiarando ciò mediante una clausola **throws** nell'interfaccia del metodo.

Si capisce quindi che la dichiarazione delle eccezioni deve essere parte integrante dell'interfaccia di ogni metodo così da rendere consapevoli agli utilizzatori di quel metodo delle eccezioni che possono essere sollevate.

Alternativamente esiste anche la possibilità di sfruttare il costrutto **finally** che permette alla fine del blocco **try-catch** di chiudere la risorsa aperta indipendentemente se l'eccezione è stata lanciata o meno. FINALLY HA UNA PRECEDENZA MOLTO FORTE E PRIORITARIA RISPETTO ALLE ALTRE ISTRUZIONI (surclassa persino un eventuale return statement nel blocco try).

# Gerarchia di eccezioni

Le eccezioni possono essere raggruppate in categorie per esempio relative alle interfacce di input output, relative al display, ecc…. Poiché le eccezioni sono rappresentate come oggetti, l'ereditarietà fornisce un meccanismo naturale per rappresentare tale raggruppamento. L'**eredità permette inoltre di creare eccezioni definite dal programmatore** come **sottoclassi di Exception**.

# Throwable e sottoclassi

E' importante notare che gli oggetti che possono essere "sollevati" da **throw** devono avere come tipo una **sottoclasse di Throwable**. Tra tali sottoclassi si trovano due classi distinte: **Error** ed **Exception**.

Le sottoclassi di **Error** rappresentano **errori gravi** che spesso non possono essere gestiti dai programmi, tant'è che è inusuale per un programma sollevare un errore di questo tipo.

Le sottoclassi di **Exception** rappresentano le comuni eccezioni di cui abbiamo trattato fino ad ora. Fra queste sottoclassi si trovano in particolare le sottoclassi di **RunTimeException** che definiscono una famiglia di eccezioni cosiddette *runtime* con regole proprie. Tali classi rappresentano quindi errori che avvengono nell'interprete di Java quindi nella Java Virtual Machine, ma che non sono così gravi da essere considerati Error. Un esempio è l'errore **NullPointerException.**

# Eccezioni e polimorfismo

Si noti che un metodo *m* definito in una sottoclasse D che ridefinisce un metodo *m* della superclasse C, può sollevare solo le *n* eccezioni dichiarate nella firma di *m* in C, oppure sottoclassi delle *n* eccezioni. Ciò vale a dire che la specifica delle eccezioni può solo "restringersi" procedendo verso le sottoclassi, ma non "allargarsi". Quindi riassumendo **le sottoclassi possono lanciare solo le eccezioni o sottoclassi di esse lanciate dalla superclasse**.
Si noti però che questa regola non vale per i costruttori in quanto non sono ereditati.