Capitolo 8

Eccezioni

Andrea Gini

Durante la normale esecuzione, un programma può andare incontro a vari problemi di funzionamento. Tali problemi, a volte, non dipendono dal codice, ma da eventi del mondo reale che non sono sotto il controllo del programmatore. Si pensi a un programma che legge un file dal disco: se durante l'esecuzione il disco si rompe, il programma andrà incontro a un fallimento. Tale fallimento non è dovuto a un errore di programmazione: si tratta semplicemente di uno di quegli incidenti che nel mondo reale possono capitare quando meno ce lo si aspetta.

Java definisce in modo rigoroso il concetto di eccezione, e prevede un apposito costrutto per favorirne la gestione. A differenza di C++, la gestione delle eccezioni in Java non è derogabile: tutte le situazioni in cui può avvenire un'eccezione devono essere gestite in modo esplicito dal programmatore.

Questo approccio rende i programmi di gran lunga più robusti, e riduce notevolmente i problemi di affidabilità e di portabilità del codice.

Errori ed eccezioni

I problemi che si possono presentare in fase di esecuzione appartengono a due categorie: errori di runtime ed eccezioni.

Gli errori di runtime si verificano quando un frammento di codice scritto correttamente si trova a dover gestire una situazione anomala, che impedisce di proseguire l'esecuzione. Si osservi il seguente assegnamento:

```
a = b / c;
```

L'istruzione è formulata correttamente e funzionerà senza problemi in quasi tutti i casi; tuttavia, se per qualche ragione la variabile `c` dovesse assumere il valore 0, l'assegnamento non potrebbe avere luogo, dal momento che la divisione per 0 non è definita. Si noti che le circostanze per cui la variabile `c` potrebbe assumere il valore 0 non dipendono necessariamente dallo sviluppatore: se il programma legge i dati da un file, per esempio, potrebbe capitare che quest'ultimo sia stato formulato in modo sbagliato dall'operatore incaricato di inserire i dati. La caratteristica principale degli errori di runtime è che essi provocano quasi sempre la chiusura irrevocabile del programma.

Le eccezioni sono situazioni anomale che interrompono un'operazione durante il suo normale svolgimento. Se un computer sta dialogando con un server tramite la rete, e sul più bello un fulmine incenerisce la linea di collegamento, il programma client dovrà affrontare una circostanza accidentale e imprevista, che in molti casi potrà essere gestita senza provocare la chiusura del programma. Chiunque navighi in Internet si sarà trovato almeno una volta nell'impossibilità di collegarsi a un sito: in casi come questo, il browser si limita a presentare un messaggio di errore a schermo, quindi si predispone a ricevere nuove direttive dall'utente.

Gestione delle eccezioni

Molti oggetti Java possono generare eccezioni durante la chiamata di un loro metodo o addirittura durante la creazione: questa esigenza è specificata dalla documentazione della classe. Il tentativo di aprire un file in lettura, per esempio, può causare una FileNotFoundException se file che si desidera aprire non esiste. Se si prova a compilare un programma che contiene un'istruzione come la seguente:

```
Reader i = new FileReader("file.txt");
```

Il compilatore segnalerà la necessità di "catturare" (catch) in modo esplicito l'eccezione FileNotFoundException:

```
C:\program.java:6: unreported exception java.io.FileNotFoundException;
 must be caught or declared to be thrown Reader i = new FileReader("file.txt");
^
1 error
```

L'esempio seguente mostra la sintassi da utilizzare nella situazione appena descritta:

```
try {
  BufferedReader i = new BufferedReader(new FileReader("text.txt"));
}
catch(FileNotFoundException fnfe) {
  System.out.println("Il file indicato non è stato trovato");
  fnfe.printStackTrace();
}
```

In questo caso, il computer *prova* (try) a eseguire l'istruzione contenuta nel primo blocco. Se durante il tentativo si verifica un'eccezione, quest'ultima viene *catturata* (catch) dal secondo blocco, che stampa a schermo un messaggio di errore e mostra lo stack di esecuzione del programma:

```
Il file indicato non è stato trovato
java.io.FileNotFoundException: file.txt (Impossibile trovare il file specificato)
        at java.io.FileInputStream.open(Native Method)
        at java.io.FileInputStream.<init>(FileInputStream.java:103)
        at java.io.FileInputStream.<init>(FileInputStream.java:66)
        at java.io.FileReader.<init>(FileReader.java:41)
        at Untitled.main(Untitled.java:7)
```

Se invece non sorgono problemi, il runtime Java ignora il contenuto nel blocco catch e prosegue nella normale esecuzione.

Costrutto try – catch – finally

Il costrutto generale per la gestione delle eccezioni ha la seguente forma:

```
try {
        istruzione1();
   istruzione2();
   ....
}
catch(Exception1 e1) {
        // gestione dell'eventuale problema nato nel blocco try
}
catch(Exception2 e2) {
        // gestione dell'eventuale problema nato nel blocco try
}
finally {
        // codice da eseguire comunque al termine del blocco try
}
```

Il blocco try contiene un insieme di istruzioni che potrebbero generare eccezioni. Generalmente, si racchiude all'interno di tale blocco un'intera procedura: se durante l'esecuzione della stessa una qualsiasi delle istruzioni genera un'eccezione, il flusso di esecuzione si interrompe e la gestione passa al blocco catch incaricato di gestire l'eccezione appena sollevata.

Il blocco catch specifica quale eccezione si desidera gestire e quali istruzioni eseguire in quella circostanza. È possibile ripetere più volte il blocco catch, in modo da permettere una gestione differenziata delle eccezioni generate dal blocco try. In Java è obbligatorio inserire un catch per ogni possibile eccezione, anche se naturalmente si può lasciare in bianco il blocco corrispondente.

La clausola finally contiene un blocco di istruzioni da eseguire comunque dopo il blocco try, sia se esso è terminato senza problemi sia nel caso abbia sollevato una qualche eccezione.

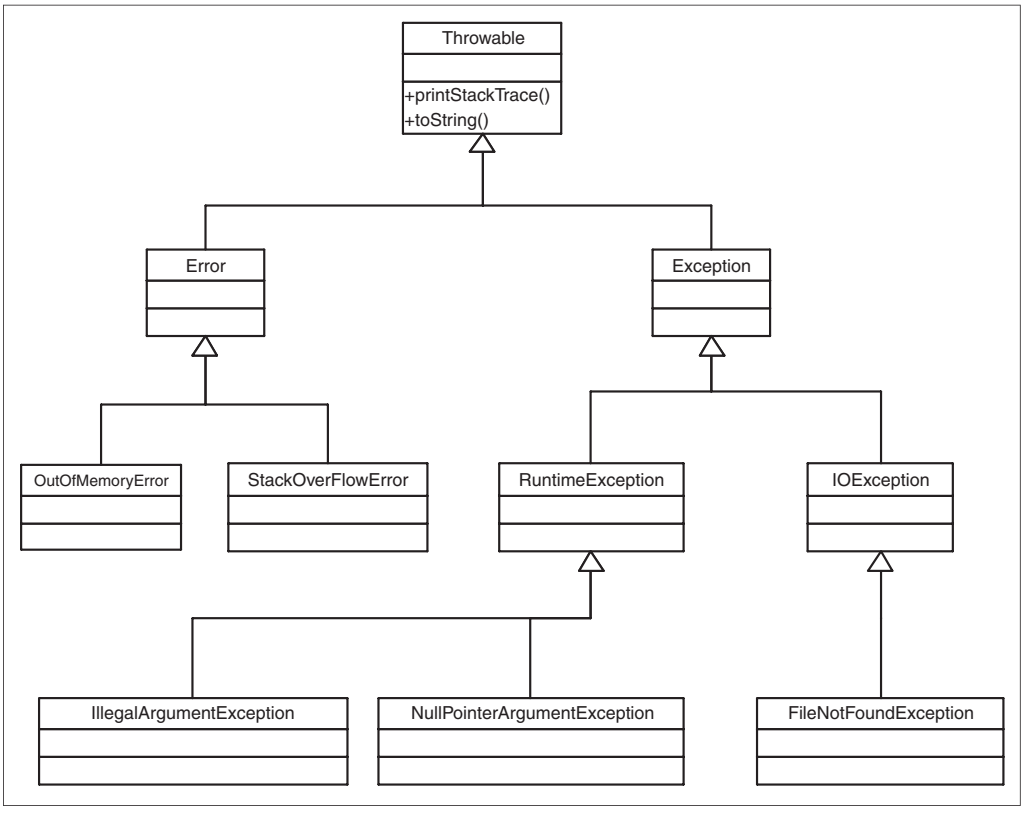
Gerarchia delle eccezioni

Le eccezioni sono oggetti e hanno una loro gerarchia. In cima a essa si trova l'oggetto Throwable, che definisce costruttori e metodi comuni a tutte le sottoclassi. Tra questi metodi, vale la pena di segnalare i seguenti, che permettono di stampare a schermo o su file informazioni diagnostiche:

- printStackTrace(): stampa su schermo lo stack di sistema, segnalando a quale punto del flusso di esecuzione l'eccezione è stata generata e come si è propagata.

- toString(): produce una stringa con le informazioni che caratterizzano l'eccezione.

Figura 9.1 – *Gerarchia delle eccezioni Java.*



La classe Throwable dà origine a una gerarchia enorme (il JDK 1.4 contiene 330 tra eccezioni ed errori). Le eccezioni possono essere suddivise in due categorie: checked e unchecked. Le prime devono obbligatoriamente essere gestite all'interno di un blocco try - catch: oltre alla generica Exception, esistono eccezioni che segnalano malfunzionamenti di input output, problemi di rete, errori nella formattazione dei dati e così via.

La seconda famiglia, che comprende gli Error, le RuntimeException e le relative sottoclassi, non obbliga il programmatore a ricorrere a una try - catch, dal momento che queste eccezioni segnalano i malfunzionamenti per i quali non è previsto recupero. Si notino OutOfMemoryError e StackOverflowError, due condizioni che tipicamente segnalano l'esaurimento delle risorse macchina e la necessità di terminare il programma.

L'organizzazione gerarchica delle eccezioni consente una gestione per categorie, secondo il principio della genericità. Per esempio, se viene generata IOException, può essere gestita sia con un'istruzione del tipo:

```
catch(IOException ioe)
```

sia con una più generica:

```
catch(Exception e)
```

È anche possibile predisporre una gestione delle eccezioni per ordine crescente di genericità. Nell'esempio seguente, i primi tre catch gestiscono eccezioni ben precise, mentre l'ultimo gestisce tutte le eccezioni che non sono state gestite dai blocchi precedenti:

```
try {
  ....
}
catch(NullPointerException npe) {}
catch(FileNotFoundException fnfe) {}
catch(IOExcetpion ioe) {}
catch(Exception e) {}
```

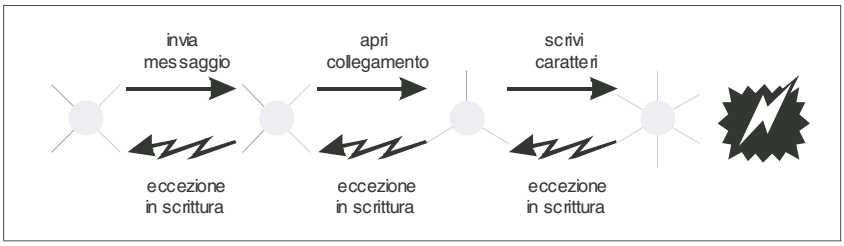
Si noti che la gestione di FileNotFoundException deve precedere quella di IOException, dato che la prima è una sottoclasse della seconda. Per la stessa ragione, la gestione di Exception deve per forza comparire in fondo all'elenco.

Propagazione: l'istruzione throws

Il costrutto try – catch permette di gestire i problemi localmente, nel punto preciso in cui sono stati generati.

Tuttavia, in un'applicazione adeguatamente stratificata, può essere preferibile fare in modo che le classi periferiche lascino rimbalzare l'eccezione verso gli oggetti chiamanti, in modo da delegare la gestione dell'eccezione alla classe che possiede la conoscenza più dettagliata del sistema.

Figura 13.2 – *Propagazione di un'eccezione in un sistema stratificato.*



L'istruzione throws, se è presente nella firma di un metodo, consente di propagare l'eccezione al metodo chiamante, in modo da delegarne a esso la gestione:

```
public void sendMessage(String message) throws IOException {
  // istruzioni che possono generare una IOException
}
```

Naturalmente, la chiamata al metodo sendMessage, definito con clausola throws, dovrà essere posta all'interno di un blocco try - catch. In alternativa, il metodo chiamante potrà a sua volta delegare la gestione dell'eccezione mediante una throws.

Lancio di eccezioni: il costrutto throw

Finora si è visto come gestire metodi che possono generare eccezioni, o in alternativa come delegare la gestione delle stesse a un metodo chiamante. Ma cosa si deve fare se si desidera *generare* in prima istanza un'eccezione? Si provi a definire un metodo per il calcolo del fattoriale: per creare una procedura robusta, è necessario segnalare un errore nel caso si provi a effettuare una chiamata con un parametro negativo, dal momento che la funzione fattoriale non è definita in questo caso. Il sistema ideale per segnalare l'irregolarità di una simile circostanza è la generazione di una IllegalArgumentException, come nell'esempio seguente:

```
public class LibreriaMatematica  {
static int fattoriale(int n) throws IllegalArgumentException {
 if(n<0)
   throw new IllegalArgumentException("Il parametro deve essere positivo");

   long f = 1;
   while ( n > 0 ) {
    f = f * n;
    n = n - 1;
   }
   return f;
 }
}
```

A questo punto, le chiamate al metodo fattoriale(int n) devono necessariamente comparire all'interno di un blocco try – catch:

```
try {
  String stringNum = JOptionPane.showInputDialog(null, "Inserisci un numero");
  int num = Integer.parseInt(stringNum);
  long res = LibreriaMatematica.fattoriale(num);
  System.out.println("Il fattoriale di " + num + " è " + res);
}
catch (IllegalArgumentException e) {
  System.out.println(e.getMessage());
}
```

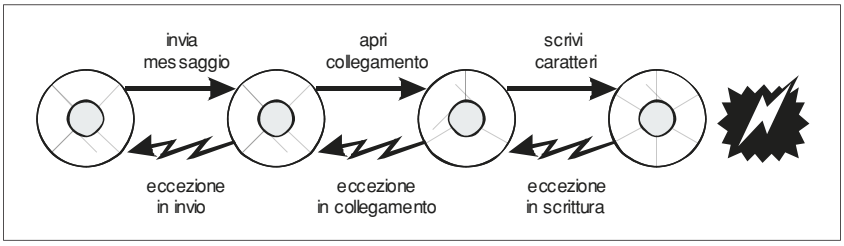
L'istruzione throw richiede come argomento un oggetto Throwable o una sua sottoclasse. È possibile utilizzare throw all'interno di un blocco catch, qualora si desideri ottenere sia la gestione locale di un'eccezione sia il suo inoltro all'oggetto chiamante:

```
public void sendMessage(String message) throws IOException {
 try {
   // istruzioni che possono generare una IOException
 }
 catch(IOException ioe) {
   System.out.println("IOException nel metodo sendMessage");      // gestione locale
   throw ioe;                                                    // inoltra l'eccezione al chiamante
 }
}
```

Catene di eccezioni

Il meccanismo di inoltro delle eccezioni descritto nel paragrafo precedente può essere ulteriormente raffinato grazie a una funzionalità introdotta a partire da Java 1.4: le eccezioni concatenate. In un sistema adeguatamente stratificato, capita che un'eccezione catturata a un certo livello sia stata generata a partire da un'altra eccezione, nata da un livello più profondo. L'uso di differenti livelli di astrazione rende spesso incomprensibile un'eccezione di livello più basso:

Figura 13.3 – *Uno scenario di generazione di eccezioni a catena.*



Per questa ragione, a partire da Java 1.4 le eccezioni permettono, in fase di creazione, di specificare una *causa*, ossia l'oggetto Throwable che ha provocato l'eccezione al livello più alto. Questo meccanismo può dar vita a vere e proprie catene di eccezioni, che forniscono una diagnostica molto dettagliata, utile a comprendere la vera natura di un problema.

Eccezioni definite dall'utente

Nonostante l'enorme varietà di eccezioni già presenti in Java, il programmatore può facilmente crearne di proprie, qualora desideri segnalare condizioni di eccezione tipiche di un proprio programma. Per creare una nuova eccezione è sufficiente dichiarare una sottoclasse di Exception (o di una qualsiasi altra eccezione esistente) e ridefinire uno o più dei seguenti costruttori:

- Exception(): crea un'eccezione.
- Exception(String message): crea un'eccezione specificando un messaggio diagnostico.
- Exception(Throwable cause): crea un'eccezione specificando la causa.
- Exception(String message, Throwable cause): crea un'eccezione specificando un messaggio diagnostico e una causa.

Ecco un esempio di eccezione personalizzata:

```
public class MyException extends Exception {

  public  MyException () {
   super();
  }
  public  MyException (String message) {
    super(message);
  }
  public Exception(String message, Throwable cause) {
    super(message, cause);
  }
  public Exception(Throwable cause) {
    super(cause);
  }
}
```

Nella maggior parte dei casi, tuttavia, è sufficiente creare una sottoclasse vuota:

```
public class MyException extends Exception {}
```

Esempio riepilogativo

Per riassumere un argomento così importante e delicato, è opportuno studiare un esempio riepilogativo. Il seguente programma solleva una IllegalArgumentException se si cerca lanciare la riga di comando con un numero di parametri diverso da uno; quindi, apre il file specificato dall'utente e ne stampa il contenuto a video. Le operazioni di apertura, chiusura e lettura del file possono generare eccezioni: alcune di queste vengono gestite direttamente dal programma, mentre altre vengono inoltrate dal metodo main al runtime Java.

```
import java.io.*;

public class ProvaEccezioni {

  public static void main(String argv[]) throws IOException {
    if(argv.length!=1)
      throw new IllegalArgumentException("Uso: java ProvaEccezioni <filename>");

    BufferedReader i = null;
    try {
      i = new BufferedReader(new FileReader(argv[0])); // throws FileNotFoundException
      String s = i.readLine();                         // throws IOException
      while(s != null) {
        System.out.println(s);
        s = i.readLine();                              // throws IOException
      }
    }
    catch(FileNotFoundException fnfe) {
      System.out.println("Il file indicato non è stato trovato");
      fnfe.printStackTrace();
    }
    catch(IOException ioe) {
      System.out.println("Errore in lettura del file");
      ioe.printStackTrace();
    }
    finally {
      i.close();                                       // throws IOException
    }
  }
}
```