Eccezioni e file

Stefano Mizzaro

Dipartimento di matematica e informatica
Università di Udine
http://www.dimi.uniud.it/~mizzaro
mizzaro@dimi.uniud.it

7 maggio 2003

Scaletta

- ***Eccezioni (cenni)***
- Flussi (stream) e file
  - File di byte
  - File di tipi primitivi
  - File ad accesso casuale

Eccezioni

- Quando si ha una situazione anomala in esecuzione (/0, lettura oltre fine file, …):
  - si sospende l'esecuzione normale
  - si getta un'eccezione: crea un'istanza di (una sottoclasse di) `java.lang.Exception`
  - si esegue codice per la gestione dell'eccezione

Gettare, catturare, rimbalzare

- *throw, catch, rethrow*
- Gettare un'eccezione
- Se viene catturata: viene gestita
- Altrimenti viene rimbalzata
  - al blocco più esterno (e al più esterno e…)
  - al chiamante (e al chiamante del chiamante…)

Gettare

- Gettare = creare un'istanza di `java.lang.Exception` (o sottoclasse)
- L'interprete Java
  - al verificarsi di una situazione anomala
- Esplicitamente dal programmatore
  - `throw <oggettoEccezione>`
  - Es.: `throw new Exception();`

Catturare

- `try/catch/finally`
- Esegue `<I>`
- Se `<I>` getta eccezione `e`, `<I>` si interrompe viene eseguita la prima `<Ii>` t.c. `e instanceof <Ei>`
- `<id>` è come un parametro formale
- `<If>` sempre eseguita alla fine

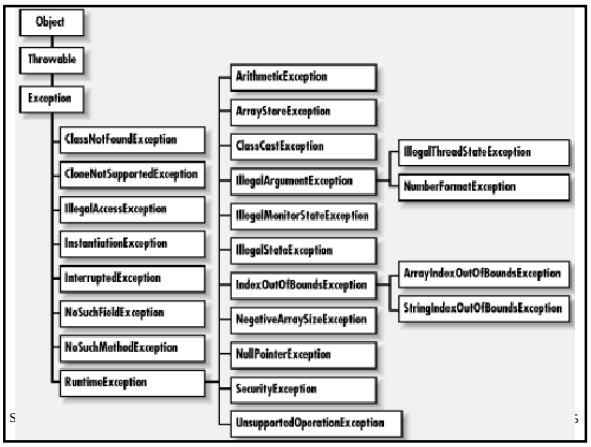
```
try {
  <I>
} catch (<E1> <id>) {
  <I1>
} catch (<E2> <id>) {
  <I2>
} ...

} catch (<En> <id>) {
  <In>
} finally {
  <If>
}
```

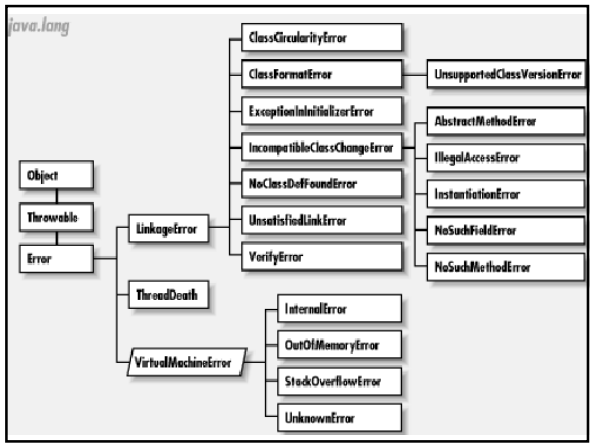
Rimbalzare (`throws`)

- Un'eccezione non catturata viene rimbalzata
  - Al blocco esterno (ricorsivamente)
  - Al metodo chiamante (ricorsivamente)
- Se `m()` rimbalza eccezioni va dichiarato:
  `public void m () throws E1, E2 {`
  - (se ***checked***, ossia non `RunTimeException`)
- Meglio catturare…

Eccezioni predefinite

- `Throwable`
  - `Exception`
  - `Error`: il programma termina
- Non catturate gli `Error`
- 100+ classi predefinite
- Vediamo la documentazione…





Eccezioni definite da noi

```
class EccezioneMia extends Exception {}
```

```
class EccezioneMia extends Exception {
  public EccezioneMia() {}
  public EccezioneMia(String s) {
    super(s);
    ...
  }
}
```


Utilità delle eccezioni

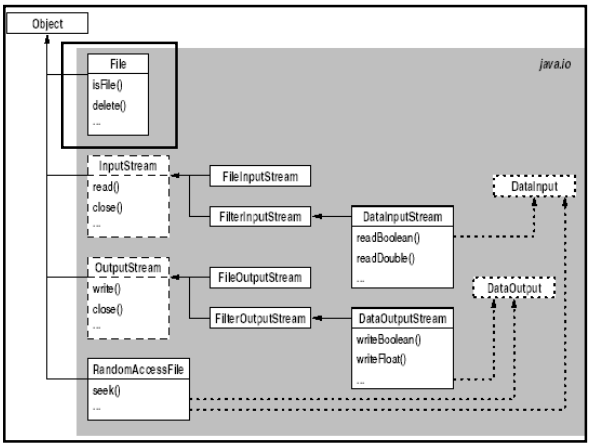
- Separare codice "normale" da situazioni anomale
- Maggiore leggibilità
- Meglio di "valore funzione =1: ⇒ errore!"
- Ne riparlerete…

Scaletta

- Eccezioni (cenni)
- ***Flussi (stream) e file***
  - File di byte
  - File di tipi primitivi
  - File ad accesso casuale

Flussi e file

- **Flusso** (stream) = sequenza di dati
  - di input: da cui leggere
  - di output: su cui scrivere
- I **file** sono flussi
- Il package `java.io` definisce parecchie (>50) classi e interfacce per la gestione dei file
- Vediamo le più semplici (e inefficienti…)



`java.io.File` (1/2)

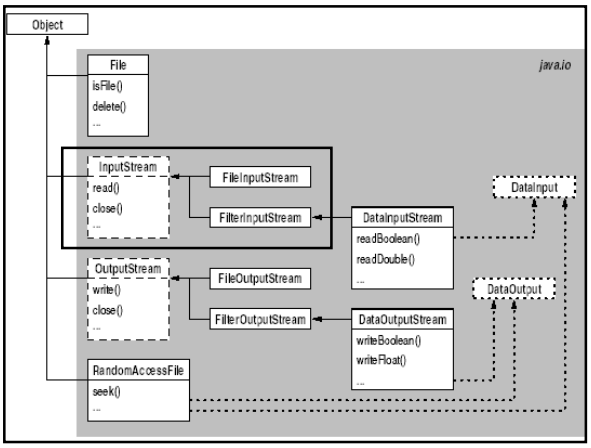
- Un'istanza di `File` rappresenta un file/pathname
- I metodi (d'istanza) di `File` vedono un file come oggetto atomico (non vedono il contenuto)
- Associare l'istanza di `File` a un pathname (con il costruttore):
  - `public File(String)`
- Verificare se un file esiste
  - `public boolean exists()`

`java.io.File` (2/2)

- Cancellare un file
  - `public boolean delete()`
- Verificare se l'istanza rappresenta un file o una directory:
  - `public boolean isFile()`
  - `public boolean isDirectory()`
- Creare un `File` vuoto:
  - `public boolean createNewFile()`

Come si lavora sul contenuto di un file

- Apertura
- Operazioni lettura/scrittura
- Chiusura



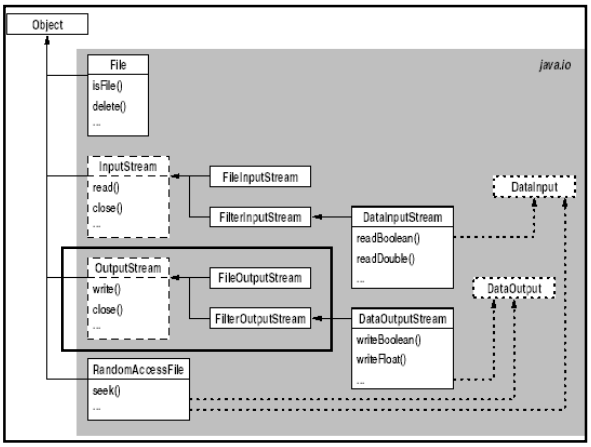
InputStream

E nei file ci sono caratteri…

- Classe astratta
  - `public abstract int read()`:
    - legge un **byte** dal flusso, ritorna il codice ASCII
    - Sovraccarico (`public int read(byte b[])`) e può leggere anche un array di byte
    - Restituisce -1 a fine file
  - `public void close()`: chiude il flusso
  - `public long skip(long n)`: salta il numero di byte specificato dal parametro

FileInputStream

- Sottoclasse di `InputStream`
- È quella da usare, i suoi oggetti sono flussi di input. Non definisce altri metodi
- (ovviamente implementa `read`)
- Quindi si può leggere il contenuto di un file, carattere per carattere



OutputStream

- Simmetria con `InputStream`
- Classe astratta
  - `public abstract void write(int b)`:
    - scrive un byte sul flusso (8 bit meno significativi)
    - sovraccarico (`public void write(byte b[])`) e può scrivere anche un array di byte alla volta.
  - `public void close()`: chiude il flusso. È importante chiudere i flussi di output per non perdere i dati non ancora salvati

FileOutputStream

- Sottoclasse di `OutputStream`
- Non definisce altri metodi
- È quella da usare, i suoi oggetti sono flussi di output.

Apertura file

- I costruttori di `FileInputStream` e `FileOutputStream` aprono un file in lettura e scrittura
- Parametro:
  - un oggetto della classe `File`:
    - ```
      FileInputStream f;
      f = new FileInputStream(new
                  File("pippo/pluto.txt"));
      ```
  - oppure una stringa (pathname del file):
    - ```
      FileInputStream f;
      f = newFileInputStream("pippo/pluto.txt");
      ```


Esempio 1: `VediFile.java`

- Scriviamo un programma per visualizzare il contenuto di un file di testo
- Uso:

```
>java VediFile pippo.txt
```


`VediFile.java` (1/2)

```
import java.io.*;
public class VediFile {
  public static void main(String args[]) {
    FileInputStream f;
    int c;
    if (args.length != 1)
      System.out.println("Uso: java VediFile <file>");
    else {
      try {
        f = new FileInputStream(args[0]);
      } catch (FileNotFoundException e) {
        System.out.println("Errore in apertura file");
        return;
      }
```



```
      try {
        c = f.read();
        while (c > 0) {
          System.out.print((char) c);
          c = f.read();
        }
      } catch (IOException e) {
        System.out.println("Errore in lettura file");
      }
      try {
        f.close();
      } catch (IOException e) {
        System.out.println("Errore in chiusura file");
      }
    }
  }
}
```


Commenti

- Cast perché `read` restituisce un `int`
- Le eccezioni vanno catturate
- `try/catch` per gestire tutte le possibili eccezioni

Es. 2 : `CopiaFile.java`

- Programma che copia un file in un altro
- Legge, carattere per carattere, da un `(File)InputStream` e…
- … scrive, carattere per carattere, su un `(File)OutputStream`
- Uso:

```
>java CopiaFile pippo.txt pluto.txt
```

CopiaFile.java (1/2)

```
import java.io.*;
public class CopiaFile {
  public static void main(String args[])
              throws FileNotFoundException,
  IOException {
    FileInputStream in;
    FileOutputStream out;
    int c;
    if (args.length != 2)
      System.out.println(
        "Uso: java CopiaFile <fileIn> <fileOut>");
    else {
```


CopiaFile.java (2/2)

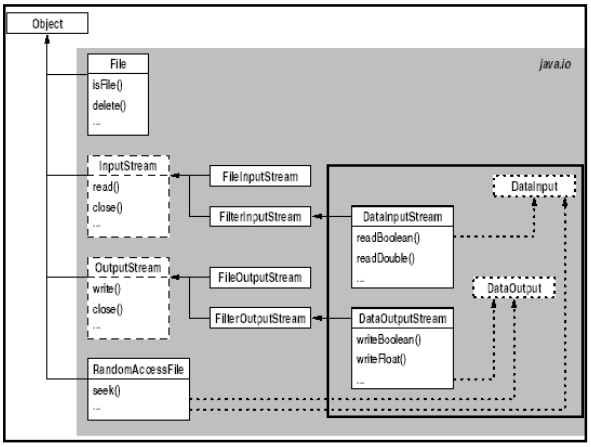
```
      in = new FileInputStream(args[0]);
      out = new FileOutputStream(args[1]);
      c = in.read();
      while (c > 0) {
        out.write((char) c);
        c = in.read();
      }
      in.close();
      out.close();
    }
  }
}
```


Commenti

- Non abbiamo catturato le eccezioni (male!)
- Il file di output viene creato dal costruttore di `FileOutputStream`
- Il corpo del ciclo ha la stessa struttura di `VediFile`

I file e i tipi primitivi

- Finora: lettura/scrittura byte per byte, carattere per carattere
- È comodo usare i tipi primitivi (leggere e scrivere `double`, `boolean`, . . .):
  - `DataInputStream` (che implementa `DataInput`)
  - `DataOutputStream` (che implementa `DataOutput`)



DataInput(Stream) e DataOutput(Stream)

- Le interfacce `DataInput` e `DataOutput` contengono metodi del tipo `readBoolean()`, `readByte()`, `writeDouble()`, `writeShort()`, . . .
- `DataInputStream` e `DataOutputStream`
  - simili alle classi viste finora
  - però i costruttori non sono sovraccarichi e vogliono come parametro un oggetto di tipo, rispettivamente, `InputStream` e `OutputStream` (non una stringa)

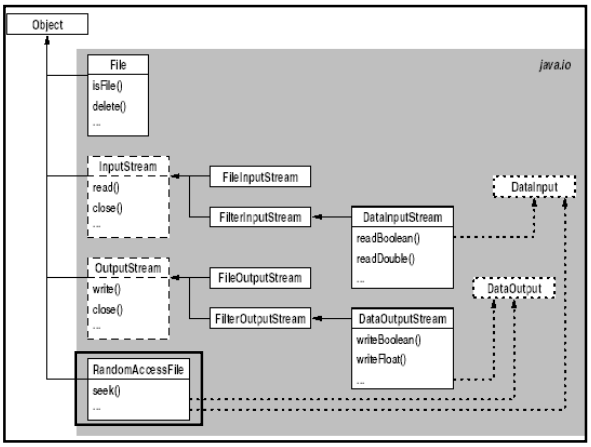
```
import java.io.*;
public class DataFile {
  public static void main (String[] args)
                                  throws IOException {
    DataOutputStream out;
    DataInputStream in;
    out = new DataOutputStream(new
                            FileOutputStream("tmp"));
    out.writeBoolean(false);
    out.writeDouble(12.34);
    out.writeChar('q');
    out.close();
    in = new DataInputStream(new FileInputStream("tmp"));
    System.out.println(in.readBoolean());
    System.out.println(in.readDouble());
    System.out.println(in.readChar());
    in.close();
  }
}
```

Commenti

- Esecuzione
- No eccezioni…

I file ad accesso casuale (diretto)

- Accesso diretto, non sequenziale
- `java.io.RandomAccessFile`
- implementa `DataInput` e `DataOutput`
- ha un metodo `seek(int)` per posizionarsi all'n-esimo byte del file (1° è zero)
- Ha anche altri metodi (tutti quelli di `DataInput` e `DataOutput`)



Esempio: lettura da file ad accesso casuale

- Scriviamo 6 caratteri su un file ad accesso casuale
- Poi li leggiamo
  - in ordine inverso
  - dal quinto al primo

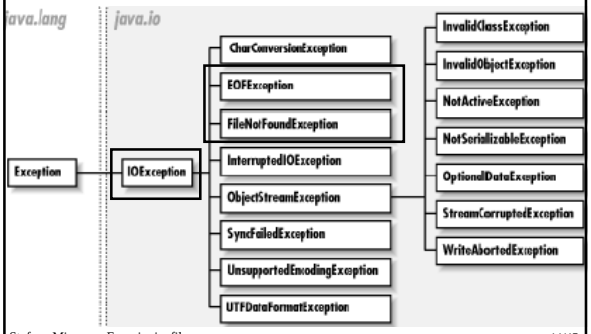
`FileAccessoCasuale.java`

```
import java.io.*;
class FileAccessoCasuale {
  public static void main (String[] args)
                        throws IOException, EOFException {
    FileOutputStream out = new FileOutputStream("tmp");
    for (char c = 'a'; c <= 'f'; c++)
      out.write(c);
    out.close();
    RandomAccessFile f = new RandomAccessFile("tmp", "r");
    for (int i = 4; i >= 0; i--) {
      f.seek(i);
      System.out.println((char)f.readByte());
    }
  }
}
```

Commenti

- Eccezioni…
- Costruttore con 2 parametri:
  - `"r"` indica lettura (read)
  - `"rw"` indica lettura & scrittura (read & write)
  - Evitare operazioni continue in memoria secondaria (es.: ordinamento di file)

Eccezioni di `java.io`



Riassunto

- Eccezioni
  - Gettare, catturare, rimbalzare
  - `try/catch`
  - `throw, throws`
- Flussi e file
  - `File`, `InputStream`, `FileInputStream`, `OutputStream`, `FileOutputStream`, `DataInputStream`, `DataOutputStream`, `RandomAccessFile`