

Gestione dei file e delle eccezioni

Lezione X

Scopo della lezione

- Introdurre le classi per la gestione dei file
- Spiegare come gestire le condizioni di errore

Laboratorio di Programmazione

X.2

La classe `File`

- Il nome è fuorviante! Rappresenta il percorso di un file o di una directory (che possono anche non esistere)
- Consiste di una sequenza di nomi separati da `File.separator` o `File.separatorChar` (ad esempio `'/'` o `'\'`)
- Fa parte del package `java.io.*`

X.3

La classe `File`

- Quando rappresenta una directory i metodi `list()` e `list(FilenameFilter)` restituiscono la lista dei file contenuti come `String[]`
- Permette di conoscere le proprietà del file
 - `canRead()/canWrite()`
 - `exists()`
 - `isDirectory()/isFile()/isHidden()`
- Leggete la descrizione di `File` sulle API

X.4

La classe File

- Scrivete un programma che legge e stampa a video l'elenco dei file contenuti in una directory (passata come argomento al metodo main della classe)
- Arricchite l'output del programma, in modo da elencare anche i diritti di lettura e scrittura sui singoli file
- Confrontate la vostra soluzione con quella proposta nelle slide seguenti

X.5

Lista dei file

```
import java.io.File;
import java.util.Date;
import prog.io.*;

class Ls {
    public static void main(String argv[]) {

        ConsoleOutputManager out = new ConsoleOutputManager();

        File f1 = new File(args[0]), f;
        String[] ls = f1.list();
```

X.6

Lista dei file

```
if (ls != null)
  for (int i = 0; i < ls.length; i++) {
    f = new File(fl, ls[i]);
    out.print(f.isDirectory() ? "d" : "-");
    out.print((f.canRead()?"r":"-")+
      (f.canWrite()?"w":"-"));
    out.print("\t"+f.length()+"\t");
    out.print(new Date(f.lastModified()));
    out.print('\t');
    out.println(f.getName());
  }
} // main()
}
```

X.7

Lista dei file

```
[16:02]cazzola@ulik:esercizi>java Ls ShapeHierarchy
-rw 945  Thu Nov 27 22:20:56 CET 2003  Rectangle.class
drw 568  Thu Nov 27 22:21:58 CET 2003  package-doc
-rw 751  Thu Nov 27 22:20:56 CET 2003  Circle.class
-rw 429  Thu Nov 27 22:20:56 CET 2003  Shape.class
-rw 597  Thu Nov 27 22:20:56 CET 2003  Square.class
-rw 2549 Thu Nov 27 22:23:12 CET 2003  ShapeHierarchy.jar
-rw 1017 Thu Nov 27 22:17:14 CET 2003  Rectangle.java
-rw 514  Thu Nov 27 22:17:14 CET 2003  Circle.java
-rw 358  Thu Nov 27 22:17:14 CET 2003  Square.java
-rw 977  Thu Nov 27 22:17:14 CET 2003  Shape.java
```

X.8

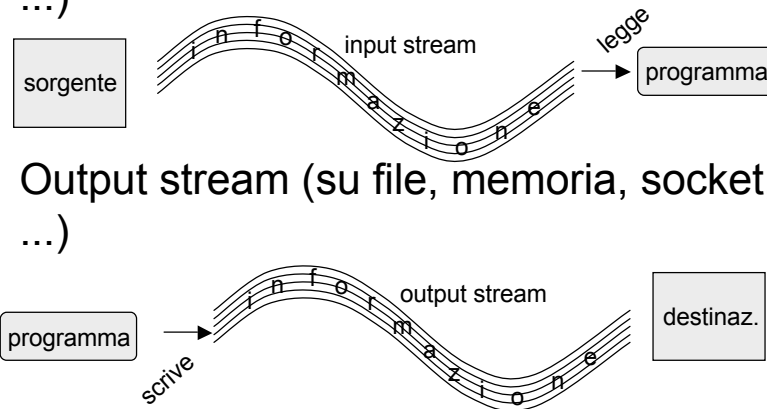
Lista dei file

- Quali funzionalità aggiuntive, rispetto a quanto richiesto, sono implementate nel programma della slide precedenti?
- Verificate i punti in cui la vostra soluzione differisce da quella proposta, e se le differenze sono sostanziali implementate anche la soluzione proposta

X.9

Stream

- Input stream (da file, memoria, socket, ...)
- Output stream (su file, memoria, socket, ...)



X.10

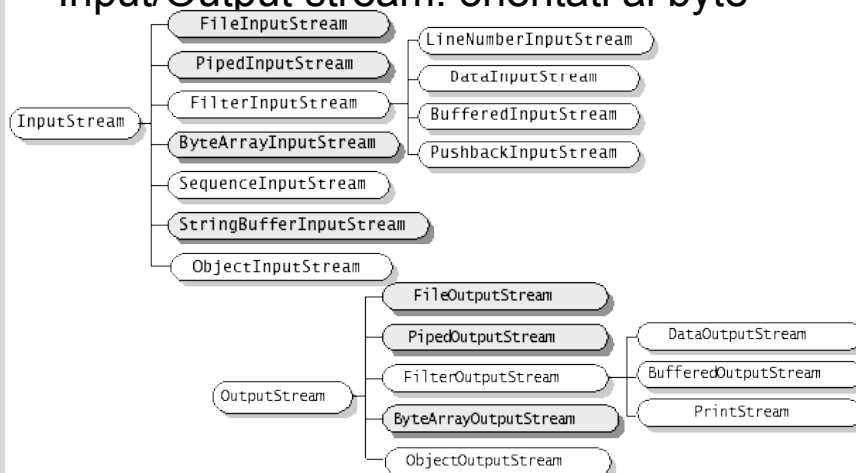
Stream

- Gli stream rappresentano dei flussi di informazioni che partono da (o arrivano a) un programma Java
- Tipicamente, all'altro lato della comunicazione si trova un file
- Le slide successive elencano le classi del package java.io che utilizzano gli stream

X.11

Byte e character stream

- Input/Output stream: orientati ai byte



X.12

- X.13

Byte e character stream

- # Laboratorio di Programmazione



Stream orientati ai caratteri

- Gli stream orientati ai caratteri
 - Trasformano i dati che devono scrivere su file in stringhe
 - Leggono stringhe da file e le convertono in un particolare tipo di dati
- Pertanto visualizzando il file utilizzato si ottiene un output leggibile “a vista”

X.15

Funzionalità delle classi

•InputStream

- `int read()`
- `int read(byte[] b)`
- `int read(byte[] b, int off, int len)`
- `void close()`

•Reader

- `int read()`
- `int read(char cbuf[])`
- `int read(char cbuf[], int off, int len)`
- `void close()`

- Che differenze ci sono tra i metodi di InputStream (orientato ai byte) e quelli di Reader (orientato ai caratteri)?

X.16

Funzionalità delle classi

•OutputStream

- void write(byte[] b)
- void write(byte[] b, int off, int len)
- int write(int b)
- void close()
- void flush()

•Writer

- void write(char cbuf[])
- void write(char cbuf[], int off, int len)
- void write(int c)
- void write(String str)
- void write(String str, int off, int len)
- void close()
- void flush()

- Che differenze ci sono tra i metodi di OutputStream e quelli di Writer?

X.17

Gestione di uno stream

- Lo stream viene aperto automaticamente quando viene creato l'oggetto
- Si leggono/scrivono i dati
- Lo stream viene chiuso esplicitamente chiamando il metodo `close()` o implicitamente dal garbage collector (GC)

X.18

Eccezioni

- Quasi tutti i metodi/costruttori possono lanciare eccezioni di I/O (IOEXception)
- Le eccezioni rappresentano gli errori che si possono verificare durante l'esecuzione
- Tratteremo questo argomento in modo approfondito tra poco
- Sarà però necessario, fin da ora, effettuare alcune piccole modifiche al codice che usiamo di solito

X.19

Esercizio

- Scrivere un programma che esegue la copia di un file (suggerimento: leggete il file sorgente carattere per carattere, e scrivete ogni carattere letto nel file di destinazione)
- Probabilmente il compilatore emetterà degli errori relativi alla gestione delle eccezioni: verificate nella slide successiva come comportarvi

X.20

Copia di un file

```
import java.io.*;

public class CopyBytes {
    public static void main(String[] args)
        throws IOException {

        FileInputStream in = new FileInputStream(args[0]);
        FileOutputStream out = new FileOutputStream(args[1]);
        int c;
        while ((c = in.read()) != -1) out.write(c);
        in.close();
        out.close();
    }
}
```

X.21

Copia di un file

```
malchiod% echo "Nel mezzo del cammin di nostra vita mi
ritrovai in una selva oscura che la diritta via era
smarrita e quanto a dir qual era  cosa dura esta selva
selvaggia ed aspra forte che nel pensier rinnova la
paura." > input
malchiod% ls
CopyBytes.class  CopyBytes.java  input
malchiod% java CopyBytes input output
malchiod% ls
CopyBytes.class  CopyBytes.java  input
output
malchiod% cat output
Nel mezzo del cammin di nostra vita mi ritrovai in una
selva oscura che la diritta via era smarrita e quanto a
dir qual era  cosa dura esta selva selvaggia ed aspra
forte che nel pensier rinnova la paura.
malchiod%
```

X.22

Eccezioni

- Quale parte del codice avete dovuto modificare per tener conto della gestione delle eccezioni

X.23

FilterStream

- Sottoclassi di `FilterInputStream` e `FilterOutputStream`: incapsulano un altro stream e ne “filtrano” i dati
 - `DataInputStream/DataOutputStream`
 - `BufferedInputStream/BufferedOutputStream`
 - `PushbackInputStream`
 - `PrintStream`

X.24

Esempio

```
import java.io.*;

class DataOut {
    public static void main(String[] args)
        throws IOException {

        DataOutputStream out = new DataOutputStream(
            new FileOutputStream("File.out"));

        out.writeDouble(3.14);
        out.writeChar('\t');
        out.writeInt(42);

        out.close();
    }
}
```

X.25

Esempio

```
import java.io.*;

class TextOut {
    public static void main(String[] args)
        throws IOException {

        PrintWriter out = new PrintWriter(new
            FileOutputStream("File.txt"));

        out.print(3.14);
        out.print('\t');
        out.print(42);

        out.close();
    }
}
```

X.26

Esempio

```
malchiod% ls
DataOut.class  DataOut.java  TextOut.class
TextOut.java
malchiod% java DataOut
malchiod% java TextOut
malchiod% ls
DataOut.class  DataOut.java  File.out
File.txt      TextOut.class  TextOut.java
malchiod% cat File.out
*@      ?°malchiod%
malchiod% cat File.txt
3.14    42malchiod%
```

X.27

Scrittura su file

- Che differenza c'è tra i due approcci alla scrittura su file implementati in `DataOut` e in `TextOut`?
- Implementate le due classi e verificate il funzionamento
- Modificatele in modo da scrivere su file espressioni di tutti i tipi di dati che conoscete

X.28

Lettura da file

- Implementate due classi `DataIn` e `TextIn` che leggono i file prodotti da `DataOut` e `TextOut` e ne stampano i contenuti a video
- Tenete conto che in questo caso dovete seguire un protocollo ben preciso: sarà necessario leggere innanzitutto un numero decimale, poi un carattere, ...

X.29

Standard Stream

- Tre campi di `System`
 - `InputStream in`
 - `PrintStream out`
 - `PrintStream err`
 - `System.out.println("ciao!");`
- Modificabili tramite tre metodi (`setIn`, `setOut`, `setErr`) della classe `System` (leggetene la documentazione sulle API)

X.30

Standard stream

- Che cosa succede se i metodi `print` e `println` vengono chiamati su `System.err` invece che su `System.out`?
- Impostate `System.out` in modo da scrivere su un file. Riuscite a scrivere un programma che emette dell'output su questo file e degli errori su video?
- Che cosa succede se anche `System.err` scrive su un file?

X.31

Lettura da standard input

- La slide seguente contiene un programma che legge da tastiera due valori interi, memorizzandoli rispettivamente in una variabile `int` e in un oggetto di `Integer`, senza ricorrere all'uso di `ConsoleInputManager`
- Analizzate il programma e capitenne il funzionamento
- Implementate il programma

X.32

Lettura da standard input

```
import java.io.*;

class In {
    public static void main(String[] args) throws
        IOException {

        BufferedReader input = new BufferedReader(new
            InputStreamReader(System.in));

        Integer objI = Integer.valueOf(input.readLine()) ;
        System.out.println(objI);
        int i = Integer.parseInt(input.readLine()) ;
        System.out.println(i);
    }
}
```

X.33

Lettura da standard input

- Scrivere classi analoghe a In che leggono da standard input dei valori con virgola decimale, dei caratteri e delle stringhe

X.34

Eccezioni

- Si definisce eccezione un errore la cui localizzazione
 - nello spazio (i.e. all'interno del codice sorgente), e/o
 - nel tempo (rispetto all'istante di esecuzione)risulta a priori difficile quando non impossibile

X.35

Esempio

```
import java.io.*;
import prog.io.*;
class EccezioneFileErrata {
    public static void main(String[] args) {
        ConsoleOutputManager video=new ConsoleOutputManager();
        BufferedReader in =
            new BufferedReader(new FileReader("elenco.txt"));
        String line = in.readLine();
        int i=1;
        while(line!=null) {
            i++;
            line = in.readLine();
        }
        video.println("Il file e' lungo " + (i-1) + " righe.");
    }
}
```

X.36

Esercizio

- Implementare il programma descritto nella slide precedente e verificare che non è possibile compilarlo
- Le slide successive riportano l'errore che dovrete ottenere quando compilate il programma, unitamente ad alcuni suggerimenti per capire come mai si verificano questi errori

X.37

Esempio

```
malchiod% javac EccezioneFileErrata.java
EccezioneFileErrata.java:8: unreported exception
    java.io.FileNotFoundException; must be caught or
    declared to be thrown
        new BufferedReader(new FileReader("elenco.txt"));
                               ^
EccezioneFileErrata.java:9: unreported exception
    java.io.IOException; must be caught or declared to be
    thrown
    String line = in.readLine();
                        ^
EccezioneFileErrata.java:13: unreported exception
    java.io.IOException; must be caught or declared to be
    thrown
    line = in.readLine();
                ^
3 errors
```

X.38

Esempio

```
import java.io.*;
import prog.io.*;
class EccezioneFileErrata {
    public static void main(String[] args) {
        ConsoleOutputManager video=new ConsoleOutputManager();
        BufferedReader in =
        new BufferedReader(new FileReader("elenco.txt"));
        String line = in.readLine();
        int i=1;
        while(line!=null) {
            i++;
            line = in.readLine();
        }
        video.println("Il file e' lungo " + (i-1) + " righe.");
    }
}
```

Cosa succede se elenco.txt non esiste?

X.39

Eccezioni

- Nel caso si verifichi una **condizione inattesa** (come la mancanza del file nell'esempio precedente), una classe Java lancia un'**eccezione**
- Il compilatore rileva le situazioni che potrebbero generare delle eccezioni e richiede al programmatore di trattarle in modo **dedicato ed esplicito**

X.40

Quindi...

```
import java.io.*;
import prog.io.*;
class EccezioneFileErrata {
    public static void main(String[] args) {
        ConsoleOutputManager video=new ConsoleOutputManager();
        BufferedReader in =
            new BufferedReader(new FileReader("elenco.txt"));
        String line = in.readLine();
        int i=1;
        while(line!=null) {
            i++;
            line = in.readLine();
        }
        video.println("Il file e' lungo " + (i-1) + " righe.");
    }
}
```

se elenco.txt non esiste
viene lanciata l'eccezione
FileNotFoundException

X.41

Analogamente...

```
import java.io.*;
import prog.io.*;
class EccezioneFileErrata {
    public static void main(String[] args) {
        ConsoleOutputManager video=new ConsoleOutputManager();
        BufferedReader in =
            new BufferedReader(new FileReader("elenco.txt"));
        String line = in.readLine();
        int i=1;
        while(line!=null) {
            i++;
            line = in.readLine();
        }
        video.println("Il file e' lungo " + i + " righe.");
    }
}
```

Queste istruzioni
possono lanciare
un'eccezione
IOException

X.42

Eccezioni

- La filosofia di base di Java prevede che un codice mal progettato non verrà mai eseguito
- Ogni volta che del codice potrebbe lanciare delle eccezioni, alternativamente
 - deve essere scritto del codice aggiuntivo per gestire le condizioni eccezionali
 - deve essere esplicitamente dichiarata la possibilità di emissione di un'eccezione

X.43

Dichiarare le eccezioni

- Quando un metodo contiene codice che potrebbe lanciare delle eccezioni, la relativa intestazione viene modificata indicando quante e quali sono queste eccezioni
- La parola chiave `throws`, inserita dopo l'elenco dei parametri formali nella dichiarazione di un metodo, dichiara quali eccezioni possono essere lanciate

X.44

Dichiarare le eccezioni

```
import java.io.*; import prog.io.*;
class EccezioneFile {
    public static void main(String[] args)
        throws FileNotFoundException, IOException {
        ConsoleOutputManager video=new ConsoleOutputManager();
        BufferedReader in =
            new BufferedReader(new FileReader("elenco.txt"));
        String line = in.readLine();
        int i=1;
        while(line!=null) {
            i++;
            line = in.readLine();
        }
        video.println("Il file e' lungo " + (i-1) + " righe.");
    }
}
```

X.45

Esercizio

- Implementare il programma descritto nella slide precedente, e verificare che sia possibile compilarlo senza che vengano emessi degli errori
- Eseguire il programma sia in presenza che in assenza del file elenco.txt che viene elaborato, e verificare che cosa succede

X.46

Dichiarare le eccezioni

```
malchiod% cat elenco.txt
riga 1
riga 2
riga 3
riga 4
riga 5
riga 6
malchiod% java EccezioneFile
Il file e' lungo 6 righe.
malchiod% rm elenco.txt
malchiod% java EccezioneFile
Exception in thread "main" java.io.FileNotFoundException:
elenco.txt (No such file or directory)
    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 EccezioneFile.main(EccezioneFile.java:7)
```

X.47

Esercizio

- Come possiamo interpretare i contenuti delle varie righe nel messaggio che viene stampato a video quando, nella slide precedente, si è verificata un'eccezione?

X.48

Risolvere le eccezioni

- In un generico linguaggio di programmazione gli approcci alla risoluzione delle eccezioni sono principalmente due
 - il programmatore deve verificare che **prima** di eseguire un'istruzione questa non dia luogo ad un'eccezione
 - il programmatore deve occuparsi di gestire le eccezioni solo **dopo** che queste sono state lanciate

X.49

Verifica delle eccezioni

```
import java.io.*;
import prog.io.*;
class EccezioneFileErrata {
    public static void main(String[] args) {
        ConsoleOutputManager video=new ConsoleOutputManager();
        BufferedReader in =
            new BufferedReader(new FileReader("elenco.txt"));
        String line = in.readLine();
        int i=1;
        while(line!=null) {
            i++;
            line = in.readLine();
        }
        video.println("Il file e' lungo " + (i-1) + " righe.");
    }
}
```

- Nel primo approccio, prima di eseguire ognuna di queste istruzioni devo verificarne la validità

X.50

Gestire le eccezioni

- Java adotta la seconda soluzione, in quanto
 - permette spesso di scrivere un unico blocco di codice che gestisca eccezioni che si possono presentare in parti diverse del codice
 - lascia che il programmatore possa separare l'implementazione del codice che gestisce le situazioni tipiche da quello che gestisce le situazioni eccezionali

X.51

Gestire le eccezioni

- La gestione delle eccezioni avviene
 - dichiarando all'interno di un metodo una **sezione critica** di codice che potrebbe lanciare un'eccezione (o più eccezioni)
 - facendo seguire questa sezione da uno o più blocchi di codice deputati a gestire le eccezioni che possono essere lanciate

X.52

Gestire le eccezioni

- Una sezione critica viene indicata racchiudendo il codice relativo in una coppia di parentesi graffe precedute dalla parola chiave `try`
- La parte di codice che gestisce un'eccezione segue questo blocco, racchiusa tra parentesi graffe e preceduta dalla parola chiave `catch` che specifica anche l'eccezione gestita

X.53

Gestione delle eccezioni

```
try {  
    Sezione critica  
} catch(Exception_1 e) {  
    Gestione dell'eccezione  
di tipo Exception_1  
}  
catch(Exception_2 e) {  
    Gestione dell'eccezione  
di tipo Exception_2  
}
```

- Se durante l'esecuzione della sezione critica non vengono lanciate eccezioni, l'esecuzione procede in modo regolare e i blocchi individuati da `catch` vengono ignorati

X.54

Gestione delle eccezioni

```

try {
    Sezione critica
    Exception_1
} catch(Exception_1 e) {
    Gestione dell'eccezione di tipo Exception_1
}
catch(Exception_2 e){
    Gestione dell'eccezione di tipo Exception_2
}
  
```

- Se viene lanciata un'eccezione di tipo `Exception_1`
 - viene interrotta l'esecuzione della sezione critica
 - viene eseguito il blocco `catch` corrispondente
 - viene ignorato il rimanente blocco `catch`

X.55

Gestione delle eccezioni

```

try {
    Sezione critica
    Exception_2
} catch(Exception_1 e) {
    Gestione dell'eccezione di tipo Exception_1
}
catch(Exception_2 e){
    Gestione dell'eccezione di tipo Exception_2
}
  
```

- Se viene lanciata un'eccezione di tipo `Exception_2`
 - viene interrotta l'esecuzione della sezione critica
 - viene ignorato il primo blocco `catch`
 - viene eseguito il blocco `catch` corrispondente a `Exception_2`

X.56

Eccezioni vs. switch

- La valutazione delle clausole catch nella gestione delle eccezioni ricorda il costrutto `switch`
- Viene eseguita solo la porzione di codice corrispondente all'eccezione lanciata, **MA**
- Non sono previste istruzioni speciali (come `break`) per chiudere i singoli blocchi `catch`

X.57

Esempio

```
import java.io.*;
import prog.io.*;

class EccezioneGestita {
    public static void main(String[] args) {
        ConsoleOutputManager video=new ConsoleOutputManager();
        try {
            BufferedReader in =
                new BufferedReader(new FileReader("elenco.txt"));
            String line = in.readLine();
            int i=1;
            while(line!=null) {
                i++;
                line = in.readLine();
            }
        }
    }
}
```

X.58

Esempio

```
video.println("Il file e' lungo " + (i-1) + "
righe.");
}
catch(FileNotFoundException e) {
    video.println("Il file elenco.txt non esiste");
}
catch(IOException e) {
    video.println("Si e' verificata un'eccezione di IO");
}
}
```

X.59

Esercizio

- Implementare il programma descritto nelle slide precedenti
- Che cosa fa questo programma?
- E' possibile eseguirlo in un contesto in cui sicuramente venga lanciata una `FileNotFoundException`? E una `IOException`?

X.60

Esempio

```
malchiod% java EccezioneGestita
Il file e' lungo 6 righe.
malchiod% rm elenco.txt
malchiod% java EccezioneGestita
Il file elenco.txt non esiste
malchiod%
```

X.61

throw VS catch

- Le parole chiave `throw` e `catch` sono complementari
 - un codice che utilizza `throw` gestisce le eccezioni che vengono lanciate all'interno dei suoi metodi **lanciandole** a sua volta al codice che lo ha chiamato
 - un codice che utilizza `catch` **cattura** le eccezioni che vengono lanciate all'interno dei suoi metodi

X.62

Eliminando un'eccezione

```
import java.io.*;
import prog.io.*;

class EccezioneMultipla {

    public static void main(String[] args) {
        ConsoleOutputManager video=new ConsoleOutputManager();

        try {

            BufferedReader in =
                new BufferedReader(new FileReader("elenco.txt"));
            String line = in.readLine();
            int i=1;
            while(line!=null) {
                i++;
                line = in.readLine();
            }
            video.println("Il file e' lungo " + (i-1) + " righe.");
        }
        catch(IOException e) {
            video.println("Si e' verificata un'eccezione di IO");
        }
    }
}
```

X.63

Eliminando un'eccezione

```
malchiod% javac EccezioneMultipla.java
malchiod% java EccezioneMultipla
Si e' verificata un'eccezione di IO
malchiod%
```

- Verificate che il programma viene compilato ugualmente
- Verificate che cosa cambia nell'esecuzione del programma e cercate di spiegarne il comportamento

X.64

Gerarchia di eccezioni

- In Java le eccezioni che vengono lanciate quando si verifica un errore sono anche esse istanze di una classe.
- Tutte le classi che si riferiscono a eccezioni sono sottoclassi della classe `Exception`
- Ereditare un'eccezione da un'altra eccezione permette di costruire una gerarchia di eccezioni

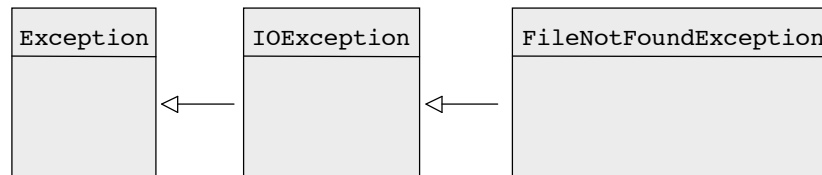
X.65

La classe `Exception`

- L'informazione più importante di una classe derivata da `Exception` è tipicamente contenuta nel suo nome
- La classe `Exception` e le sue derivate non implementano metodi particolari
 - `toString()` descrive l'eccezione
 - `printStackTrace()` stampa l'eccezione e la sequenza delle chiamate dei metodi che hanno portato a generarla

X.66

Esempio



- Siccome `FileNotFoundException` è interpretabile anche come `IOException`, la clausola `catch` del codice precedente la intercetta.

X.67

Gerarchia delle eccezioni

```
try {
}
catch(IOException e) {
    video.println("Si e' verificata un'eccezione di IO");
}
catch(FileNotFoundException e) {
    video.println("Il file elenco.txt non esiste");
}
```

- `FileNotFoundException` verrebbe sempre gestita dalla prima clausola `catch` e la seconda sarebbe inutile
- Il compilatore riconosce questa situazione ed emette un errore

X.68

Esempio

```
import java.io.*;
import prog.io.*;

class EccezioneDoppia {
    public static void main(String[] args) {
        ConsoleOutputManager video=new ConsoleOutputManager();
        try {
            BufferedReader in =
                new BufferedReader(new FileReader("elenco.txt"));
            String line = in.readLine();
            int i=1;
            while(line!=null) {
                i++;
                line = in.readLine();
            }
        }
```

X.69

Esempio

```
        video.println("Il file e' lungo " + (i-1) + "
        righe.");
    }
    catch(IOException e) {
        video.println("Si e' verificata un'eccezione di IO");
    }
    catch(FileNotFoundException e) {
        video.println("Il file elenco.txt non esiste");
    }
    }
}
```

X.70

Esempio

```
malchiod% javac EccezioneDoppia.java
EccezioneDoppia.java:21: exception
    java.io.FileNotFoundException has already been caught
    catch(FileNotFoundException e) {
    ^
1 error
malchiod%
```

- Verificare che il programma descritto non può essere compilato

X.71

Costruire nuove eccezioni

- Estendendo la classe `Exception` o una delle classi derivate da essa è possibile costruire nuove eccezioni
- All'interno del codice è possibile lanciare eccezioni (esistenti o create appositamente) tramite il comando `throw`

X.72

MyException.java

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

X.73

TestMyException.java

```
import prog.io.*;  
  
class TestMyException {  
    public static void main(String args[]) {  
        ConsoleOutputManager video=new ConsoleOutputManager();  
        try {  
            MyException m = new MyException();  
            throw m;  
        } catch(MyException e) {  
            video.println(e.toString());  
            e.printStackTrace();  
        }  
    }  
}
```

X.74

TestMyException

```
malchiod% java TestMyException
MyException
MyException
    at TestMyException.main(TestMyException.java:7)
malchiod%
```

X.75

Esercizio

- Implementare le classi `MyException` e `TestMyException` e verificare il modo in cui funzionano
- Modificare `TestMyException` in modo che lanci un'eccezione diversa da quelle viste in questa lezione (suggerimento: pensare agli errori che si sono verificati, nelle scorse lezioni, durante l'esecuzione di una classe)

X.76

TestStack.java

```
class TestStack {  
    public static void f() throws MyException {  
        throw new MyException();  
    }  
    public static void g() throws MyException {  
        f();  
        throw new MyException();  
    }  
}
```

X.77

TestStack.java

```
public static void main(String args[]) {  
    try {  
        f();  
    } catch(MyException e) {  
        e.printStackTrace();  
    }  
    try {  
        g();  
    } catch(MyException e) {  
        e.printStackTrace();  
    }  
}
```

X.78

TestStack

```
malchiod% java TestStack
MyException
    at TestStack.f(TestStack.java:3)
    at TestStack.main(TestStack.java:11)
MyException
    at TestStack.f(TestStack.java:3)
    at TestStack.g(TestStack.java:6)
    at TestStack.main(TestStack.java:16)
malchiod%
```

X.79

Esercizio

- Implementare il programma descritto nelle slide precedenti, compilarlo ed eseguirlo
- Interpretare il contenuto dei messaggi stampati a video quando si verificano le eccezioni

X.80

Runtime.java

```
class Runtime {  
    public static void main(String[] args) {  
        int i, j;  
        j=0;  
        i=3/j;  
    }  
}
```

X.81

Runtime.java

```
malchiod% javac Runtime.java  
malchiod% java Runtime  
Exception in thread "main"  
    java.lang.ArithmeticException: / by zero  
        at Runtime.main(Runtime.java:5)  
malchiod%
```

- Implementare la classe Runtime e verificare che, nonostante non venga dichiarata la possibilità di lanciare `ArithmeticException` il compilatore non emetta errore. Perché?

X.82

RuntimeException

- Eccezioni come `NullPointerException`, `ArithmeticException` e `ArrayIndexOutOfBoundsException`, che riguardano tipicamente errori di programmazione, sono raggruppate nella superclasse `RuntimeException` che non deve essere dichiarata dal programmatore

X.83

Runtime Exception

- Queste eccezioni descrivono dei tipici errori che si possono verificare mentre si sviluppa il codice, quando questo contiene degli errori che saranno rimossi nella versione definitiva
- Per questo motivo non è necessario dichiarare le eccezioni corrispondenti tramite `throws`

X.84

Finally

```
class Switch {
    boolean state = false;
    void on() {state = true; }
    void off() {state = false; }
}

class OnOffException1 extends Exception {}
class OnOffException2 extends Exception {}
```

X.85

Finally

```
class TestSwitch {
    static Switch sw = new Switch();
    static void f() throws OnOffException1, OnOffException2 {}
    public static void main(String[] args) {
        try {
            sw.on();
            // Codice che può generare eccezioni...
            f();
            sw.off();
        } catch (OnOffException1 e) {
            sw.off(); // altro codice...
        } catch (OnOffException2 e) {
            sw.off(); // altro codice...
        }
    }
}
```

Se si verificano queste eccezioni l'interruttore viene spento

Cosa succede se si verifica un altro tipo di eccezione?

X.86

Finally

- In casi come questo risulta utile poter eseguire del codice indipendentemente dalla generazione di una qualsiasi eccezione nel blocco `try`
- Questo è possibile accodando ai vari blocchi `catch` un blocco `finally`

X.87

Finally

```
class TestSwitch {
    static Switch sw = new Switch();
    static void f() throws OnOffException1, OnOffException2 {}
    public static void main(String[] args) {
        try {
            sw.on();
            // Codice che può generare eccezioni...
            f();
        } catch (OnOffException1 e) {
            // altro codice...
        } catch (OnOffException2 e) {
            // altro codice...
        } finally {
            sw.off();
        }
    }
}
```

X.88

LetturaDati.java

```
import prog.io.*;

class EndOfDataException extends Exception {
    public EndOfDataException() {
    }
}

class LetturaDati {
    public static void main(String[] args) {
        ConsoleOutputManager out = new ConsoleOutputManager();
        ConsoleInputManager in = new ConsoleInputManager();
        int somma = 0, dato;
```

X.89

LetturaDati.java

```
try {
    while(true) {
        out.println("Inserisci un intero (0 per uscire)");
        dato = in.readInt();
        if(dato != 0)
            somma += dato;
        else
            throw new EndOfDataException();
    }
} catch(EndOfDataException e) {
    out.println("Fine immisione dati");
    out.println("La somma dei numeri immessi e' " + somma);
}
}
```

X.90

Esercizio

- Implementare il programma descritto nelle slide precedenti

X.91