Dipartimento di Informatica e Sistemistica
Antonio Ruberti

“Sapienza” Università di Roma

Eccezioni

Corso di Tecniche di Programmazione

Laurea in Ingegneria Informatica

(Canale di Ingegneria delle Reti e dei Sistemi Informatici - Polo di Rieti)

Anno Accademico 2007/2008

Prof. Paolo Romano

Si ringrazia il Prof. Enrico Denti per aver reso
disponibile il proprio materiale didattico sul quale si basano queste slides

SITUAZIONI CRITICHE

- **Nei sistemi vi sono *situazioni critiche* che è *prevedibile* possano *a volte* causare *errori***
  - **l'apertura di un file potrebbe fallire**
  - **una connessione in rete potrebbe non riuscire**
  - **...**
- **Inoltre, in un sistema a oggetti, la *costruzione di un oggetto* potrebbe risultare *impossibile* in presenza di *parametri errati o assurdi***
  - **triangoli che non rispettano la condizione di esistenza**
  - **distanze negative**
  - **...**

L' APPROCCIO TRADIZIONALE

- L'approccio tradizionale consiste nell'*inserire controlli* per *intercettare a priori* l'errore, ma è un modo di procedere *insoddisfacente*
  - non è facile prevedere tutte le configurazioni che potrebbero produrre l'errore
  - "gestire" l'errore spesso significa solo stampare a video un messaggio
- Tale approccio è inoltre *inutile* nel caso della *costruzione di oggetti*
  - è un processo già in corso che non si può fermare
  - se anche si scopre il problema, *poi che si fa??*

IL CONCETTO DI ECCEZIONE

Java introduce il concetto di *eccezione:*

- come modo per sentire un *allarme* lanciato da una operazione "in crisi"
  - anziché *tentare di prevedere* l'errore, *si tenta di eseguire l'azione* in un *blocco controllato*
  - in caso di errore *il sistema lancia una eccezione*
  - l'eccezione verrà poi *catturata* da un *gestore di errore* che prenderà le contromisure del caso
- come modo per *scatenare noi un allarme* in presenza di una situazione ritenuta *critica*
  - caso particolare: costruzione di oggetti "assurdi"
  - la *costruzione* dell'oggetto viene *interrotta*

INTERCETTARE ECCEZIONI: SINTASSI

```
try {
   /* operazione critica che può
          lanciare eccezioni */
}
catch (Exception e) {
   /* gestione dell’eccezione */
}
// se tutto va bene, si prosegue qui
```

Se l’operazione lancia *diversi tipi* di eccezione in risposta a diversi tipi di errore, *più blocchi* **`catch`** (uno per ogni tipo di eccezione) possono seguire lo stesso blocco **`try`**.

FLUSSO DI ESECUZIONE

- Se l'operazione critica *ha successo*, nessuno dei blocchi `catch` viene eseguito.
- Se l'operazione critica **lancia un'eccezione**, l'esecuzione prosegue nel **blocco `catch`** appropriato.
- È possibile specificare **un blocco `finally` opzionale** *da eseguirsi comunque alla fine*, indipendentemente dal fatto che si sia eseguito il blocco `try` o un blocco `catch`.

```
try {
   ...
}
catch (IOException e){
   ...
}
catch (Exception e2) {
   ...
}
...
finally {
 // operazioni finali
}
```

ESEMPIO 1: APERTURA DI UN FILE

- In Java, l'apertura di un file di testo comporta la costruzione di un oggetto `FileReader`
  - in C#, `StreamReader`, con piccole differenze sintattiche
- L'operazione è critica, perché il file *può non esistere:* in tal caso il costruttore scatena una *`FileNotFoundException`*
- Perciò, l'apertura del file dovrà avvenire in un *blocco controllato* *`try / catch`*

```
try {
  FileReader f = new FileReader("info.txt");
}
catch(FileNotFoundException e){
  ... // gestione eccezione
}
```

ESEMPIO 2: CONVERSIONE STRINGA / NUMERO

- In Java, la conversione stringa / numero intero è svolta dalla funzione statica

```
int Integer.parseInt(String s)
```

NB: in C#, analoga funzionalità è fornita da `int.Parse(string s)`

- L'operazione è critica, perché la stringa *potrebbe non contenere la rappresentazione di un intero:* in tal caso parte una ***NumberFormatException*** *(C#: **FormatException**)*

```
try {
   int a = Integer.parseInt(stringa);
}
catch(NumberFormatException e){ ... }
```

ECCEZIONI DI DIVERSA GRAVITÀ

- **Non tutte le eccezioni hanno la stessa importanza**
  - **alcune sono *realmente critiche: non è pensabile proseguire senza gestirle* (esempio: l'apertura di un file)**
  - **altre appaiono invece *meno gravi* (esempio: la conversione stringa / numero): obbligare a gestirle sarebbe *gravoso***
- **Perciò Java distingue fra eccezioni *da controllare* ed eccezioni *non necessariamente controllate***
  - **molte *non sono da controllare obbligatoriamente (tutte quelle che derivano da* `RuntimeException`*)***
  - **le più serie invece sono *da controllare per forza***
- **C# non fa questa distinzione:**
  - **le eccezioni C# *non sono mai da controllare per forza***

ECCEZIONI CONTROLLATE e NON CONTROLLATE

- Un’operazione che possa scatenare una eccezione *da controllare per forza* deve essere eseguita dentro un `try`
  - il compilatore Java verificherà il rispetto di questa regola
  - (..che ovviamente non si applica in C#)
- Viceversa, un'operazione che possa scatenare una eccezione *non obbligatoriamente controllata* può comparire anche fuori da un blocco `try`...
  - … MA in tal caso è è responsabilità del progettista assicurarsi che essa non raggiunga il main
- Se un'eccezione incontrollata raggiunge il `main`, *il programma abortisce.*

ESEMPIO 1 con `try/catch`

L'eccezione è *da controllare per forza:* il `try/catch` in Java è indispensabile, altrimenti non si compila

```
import java.io.*;
class Esempio1 {
 public static void main(String[] args){
      try { FileReader f = new FileReader(args[0]); }
      catch(NullPointerException e){
            System.err.println("File " + args[0] + " not
found");
            System.exit(1); // esce con indicazione
d'errore
      }
 }
}
```

ESEMPIO DI ESECUZIONE (CASO FILE ASSENTE)

```
> java Esempio2 info.txt
File info.txt not found
```

ESEMPIO 1 senza `try/catch`

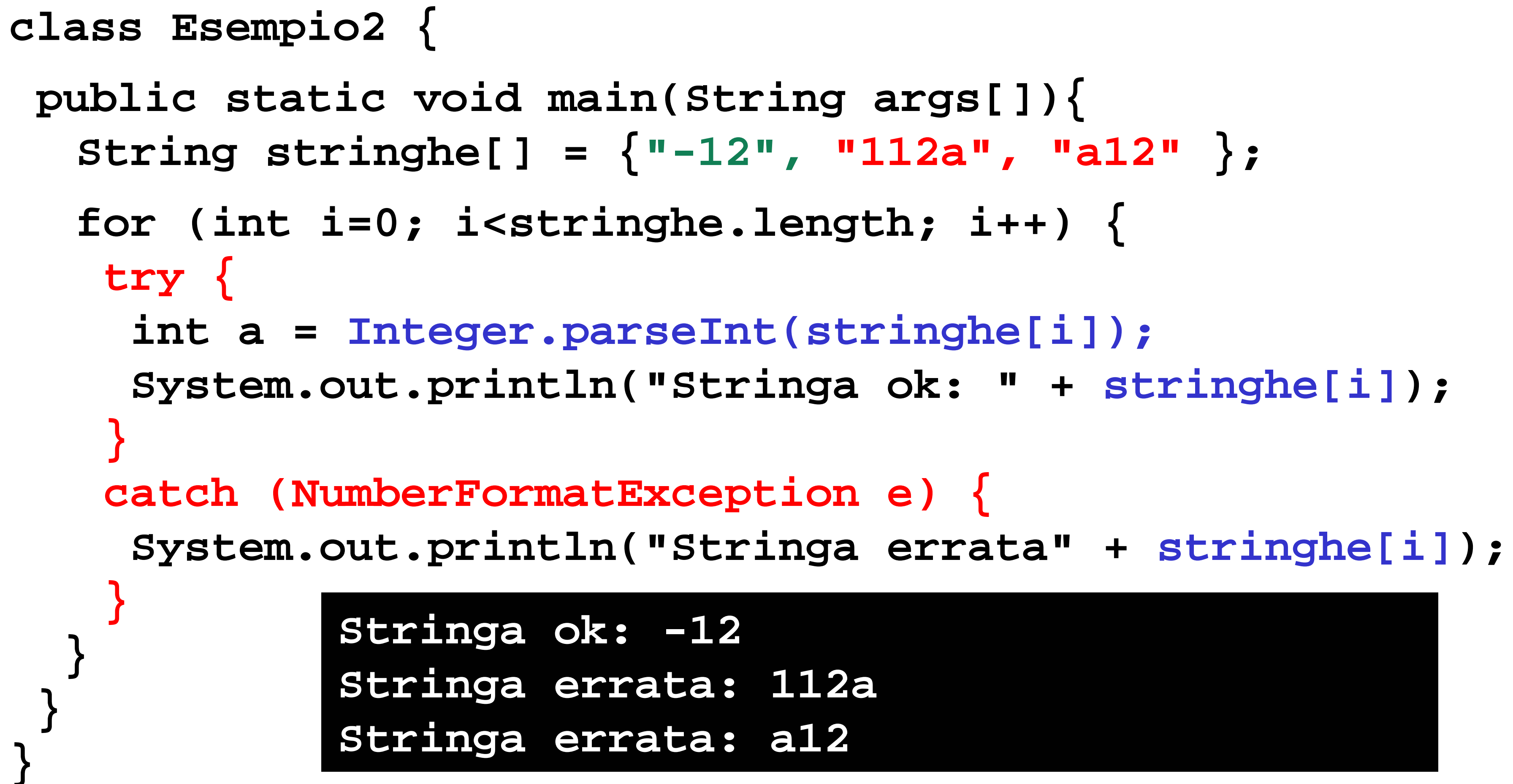
Se dimentichiamo il blocco `try` / `catch`, il compilatore Java segnala errore e impedisce di proseguire:

```
import java.io.FileReader;
class Esempio1 {
 public static void main(String[] args){
       FileReader f = new FileReader(args[0]); //
ERRATO!!
 }
}
```

```
Esempio1.java:4:
unreported exception java.io.FileNotFoundException;
must be caught or declared to be thrown
             FileReader f = new FileReader(args[0]);
                                 ^1 error
```

ESEMPIO 2 con `try/catch`

Questa eccezione *non è obbligatoriamente controllata,* ergo il blocco `try/catch` è <u>opzionale</u>



```
class Esempio2 {
 public static void main(String args[]){
   String stringhe[] = {"-12", "112a", "a12" };
   for (int i=0; i<stringhe.length; i++) {
    try {
     int a = Integer.parseInt(stringhe[i]);
     System.out.println("Stringa ok: " + stringhe[i]);
    }
    catch (NumberFormatException e) {
     System.out.println("Stringa errata" + stringhe[i]);
    }
   }
  }
}
```

```
Stringa ok: -12
Stringa errata: 112a
Stringa errata: a12
```

ESEMPIO 2 senza `try/catch`

Se omettiamo il blocco `try / catch`, il compilatore non protesta, MA si rischia errore a runtime:

```
class Esempio2 {
 public static void main(String args[]){
   String stringhe[] = {"-12", "112a", "a12" };
   for (int i=0; i<stringhe.length; i++) {
     int a = Integer.parseInt(stringhe[i]);
     System.out.println("Stringa ok: " + stringhe[i]);
   }
  }
}
```

Alla 2ª iterazione, l'eccezione raggiunge il main, facendolo abortire.

```
Stringa ok: -12
Exception in thread "main" java.lang.NumberFormatException: For input string: "112a"
 at java.lang.NumberFormatException.forInputString(Unknown Source)
 at java.lang.Integer.parseInt(Unknown Source)
 at java.lang.Integer.parseInt(Unknown Source)
 at EsempioEccezioneSenzaTryCatch.main(EsempioSenzaTryCatch.java:5)
```

ECCEZIONI: uno sguardo all'interno

- **Una eccezione è *un oggetto,* istanza di `Exception` o di una sua classe derivata**
  - **in quanto tale, può *contenere dati* o *definire metodi.***
- **Tutte le eccezioni definiscono due costruttori:**
  - **uno di default**
  - **uno con parametro stringa, che rappresenta il messaggio d'errore associato**
- **Tutte le eccezioni definiscono un metodo `getMessage`**
  - **restituisce il messaggio d'errore incapsulato**
- **Alcune eccezioni definiscono anche *campi dati***
  - `bytesTransferred` in `InterruptedIOException`

RILANCIO DI ECCEZIONI

- **Tavolta capita di *non saper cosa fare di una eccezione,* perché nel metodo dove si verifica l'errore *non si hanno informazioni sufficienti per gestirla.***
- **In tal caso, non avrebbe senso mettere nel blocco `catch` qualcosa di *"casuale"*: meglio *delegare la gestione del problema* a qualcun altro, che possa farlo meglio.**
- **A questo fine, *un metodo può rilanciare un'eccezione all'esterno di esso, rinunciando a gestirla direttamente***
  - **in tal caso, all'interno di un tale metodo non ci sarà alcun blocco `try/catch`**
- **<u>In Java</u> si richiede che *la signature del metodo avverta* che quel metodo può causare un'eccezione → *dichiarazione `throws`***

RILANCIO DI ECCEZIONI: `throws`

Ad esempio, un metodo che apra un file può decidere di *far gestire* `FileNotFoundException` *all'esterno,* ma a tal fine deve *avvisare del pericolo* nella propria signature:

```
public void leggiFile(String filename)
   throws FileNotFoundException {
      FileReader f = new
FileReader(filename);
      ...
}
```

- **Può lanciare un'eccezione → richiederebbe `try/catch`**
- **MA opta per rilanciarne la gestione all'esterno: ergo, *deve avvisare i clienti del pericolo* → clausola `throws`**

LANCIO DELIBERATO di ECCEZIONI

A volte è necessario *generare appositamente eccezioni,* per *lanciare un "nostro" allarme*

- **ad esempio, un metodo potrebbe richiedere, per funzionare, un parametro *non negativo* : che fare se gli viene passato -5 ?**
- **oppure, il costruttore potrebbe accorgersi che, con i parametri dati, risulta *impossibile costruire un oggetto : che fare?***

Soluzione:

- **si *crea esplicitamente* un *oggetto eccezione* del tipo appropriato a rappresentare l'accaduto**
- **e lo si lancia fuori con l'istruzione `throw`**

LANCIO DI UNA ECCEZIONE: `throw`

```
public int dimezza(int numeroPari)
      throws IllegalArgumentException {
```

Può lanciare una eccezione, quindi deve avvertire del rischio

```
  if (x%2==1)
    throw new IllegalArgumentException();
  else return numeroPari/2;
}
```

Per sua esplicita scelta, lancia una eccezione se il valore del parametro non è pari, in modo da avvisare il cliente della situazione eccezionale che si è presentata.

DEFINIRE NUOVI TIPI DI ECCEZIONI

Nell'esempio di poco fa, l'eccezione lanciata dal metodo è stata scelta fra quelle predefinite; non sempre però esse rappresentano bene la situazione particolare che vogliamo segnalare.

È possibile definire *nuovi tipi di eccezione,* che rappresentino nostri "tipi di errore"

```
class NumberTooBigException
                  extends IllegalArgumentException {
 public NumberTooBigException() { super(); }
 public NumberTooBigException(String s){ super(s); }
}
```

ESEMPIO D'USO

Questo metodo lancia un'eccezione del tipo ***"numero troppo grande"*** **se il risultato del calcolo è superiore a cento:**

```
public int numeroDelLotto(int num)
     throws NumberTooBigException {
  int x = num * num;
  if (x>90)
    throw new NumberTooBigException();
  else return x;
}
```

Lanciandola, si segnala un errore logico: in condizioni normali, infatti, il risultato non dovrebbe mai superare 90.

UN ALTRO ESEMPIO: IL TRIANGOLO

Condizione di esistenza

- ogni lato dev'essere minore della somma degli altri due
- il costruttore può verificarla: *se è violata, può lanciare un'eccezione* per bloccare la costruzione

```
public Triangolo(double a, double b, double c)
 throws ImpossibleTriangleException {
  if (a>=b+c || b>=a+c || c>=a+b)
    throw new ImpossibleTriangleException(
     "lati assurdi: "+a+", "+b+", "+c);
  else { // situazione normale
      this.a=a; this.b=b; this.c=c;
  }
}
```

Eccezione definita da noi

TRIANGOLO IMPOSSIBILE: L'ECCEZIONE

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

Estende `Exception` → eccezione da controllare

- il cliente dovrà usare `try / catch`, altrimenti avrà errore di compilazione all'atto della costruzione del triangolo
- È proprio quello che si vuole: un triangolo assurdo non deve poter essere costruito!

TRIANGOLO IMPOSSIBILE: UN MAIN

```
public static void main(String[] args){
 Triangolo[] trg = new Triangolo[10];
 double[][] dati = {{5,5,6}, {1,1,3}, {3,3,3}};
 // il secondo triangolo è impossibile!
 for (int i=0; i<dati.length; i++) {
      try {
            trg[i] = new Triangolo(
                  dati[i][0],dati[i][1], dati[i][2]);
        System.out.println(i+ ") " + trg[i]);
      }
      catch (ImpossibleTriangleException e) {
        System.out.println(e); // non c'è nulla da fare!
      }
 }
}
```

```
0) Triangolo di area 12.0 e perimetro 16.0
1) ImpossibleTriangleException: lati assurdi: 1.0, 1.0, 3.0
2) Triangolo di area 3.897114317029974 e perimetro 9.0
```