Programmazione ad oggetti

Gestione degli errori

A.A. 2022/2023

Docente: Prof. Salvatore D'Angelo
Email: salvatore.dangelo@unicampania.it

Università degli Studi della Campania *Luigi Vanvitelli*
Dipartimento di Ingegneria

Considerazioni

- C ed altri linguaggi presentano molti schemi per la gestione degli errori che finiscono con l'essere solo convenzioni e non parte del linguaggio.
- Tipicamente si usa ritornare un valore o settare un flag e si suppone che il destinatario controllerà per verificare che l'operazione effettuata sia andata a buon fine.
- Tuttavia quando un programmatore utilizza una libreria è propenso a credere che il suo codice sia immune da errori e non effettua tali controlli

Quando gestire gli errori

- Il momento ideale in cui catturare un errore è a tempo di compilazione, prima cioè di eseguire il programma.
- Non tutti gli errori possono essere individuati a tempo di compilazione …
- Il resto dei problemi deve essere gestito a tempo di esecuzione.
- Alcuni formalismi consentono a chi ha originato l'errore di passare le informazione appropriate a chi sarà in grado di gestire le difficoltà opportunamente

Eccezione

- La parola *“**eccezione**”* vuole dire: *“**io non mi occupo di ciò”.***
- Nel punto in cui si verifica il problema potrebbe non essere noto come gestirlo.
- Ciò che si può fare è fermare l’esecuzione e delegare ad altri il da farsi.
- Ad un livello più alto potrebbe disporsi di informazioni che rendono possibile prendere decisioni (come si fa in una gerarchia di comando).

Altri vantaggi

- Un altro vantaggio significativo delle eccezioni è la produzione di codice ordinato
- È possibile gestire l'eccezione in un sol punto del programma, nell' *exception handler*.
- La gestione delle eccezioni è l'unico metodo ufficiale utilizzato da Java per gli errori.

Eccezioni

- Una condizione di eccezione è un problema che impedisce la continuazione del metodo o dello scope in cui viene generata.
- Non è possibile continuare l'esecuzione poiché non si hanno le informazioni necessarie a prendere provvedimenti nel contesto di esecuzione corrente.
- L'unico provvedimento possibile è saltare fuori dal contesto in questione per delegare la gestione a più alto livello.
- Questo è ciò che succede quando si genera (throw) un'eccezione

Esempio: divisione

- Un semplice esempio di eccezione è la divisione per zero.
- Chi sta effettuando una divisione deve controllare che il denominatore sia diverso da zero.
- Chi implementa la divisione può non sapere cosa fare se il denominatore è zero perché non conosce il contesto.
- In tal caso tutto ciò che può fare è generare un'eccezione piuttosto che continuare .

Cercare un eccezione

- Quando si genera un eccezione accadono diverse cose.
- Prima di tutto viene creato un oggetto Eccezione come un qualsiasi oggetto Java (new Exception()).
- Quindi viene fermato il percorso di esecuzione (quello che non può continuare) e il riferimento all'oggetto eccezione creato viene espulso dal contesto corrente.
- A questo punto il meccanismo di gestione delle eccezioni comincia a cercare un punto dal quale riprendere l'esecuzione del programma.
- Tale punto è detto *exception handler*, il cui compito è recuperare lo stato corretto del problema così che il programma può percorrere un'altra via o continuare.

La pratica

- È possibile inviare una informazione di errore ad un contesto più vasto creando un oggetto che contenga l'informazione e "gettandolo" (throwing) fuori dal contesto corrente
- *throwing an exception* si traduce nel sequente codice:

```
if(t == null)
throw new NullPointerException();
```

Questo codice genera un eccezione che consente di scaricare la responsabilità di gestire la condizione di errore

- Esistono due costruttori per tutte le eccezioni standard
  - Il costruttore di default;
  - Il secondo con un argomento che descrive l'eccezione.

throw new NullPointerException("t = null");

- Questa informazione può essere recuperata da chi gestirà l'eccezione in vari modi

Guarded region

- Quando si genera una eccezione si assume implicitamente che qualcuno la gestirà ad un livello più alto.
- In Java è proprio il compilatore che si occupa di far ciò, ovvero costringe il programmatore a gestire l'eccezione.
- Prima di spiegare come introduciamo il concetto di *guarded region*

Catturare un eccezione

- Prevediamo due possibilità:
  - Il nostro stesso codice genera un eccezione: usciamo dal metodo.
  - Un metodo da noi invocato genera un eccezione
- Il blocco **try {...}** cattura un'eccezione
- Se il l'eccezione viene generata all'interno di un blocco try{...} riusciamo ad evitare l'uscita dal metodo e catturiamo l'eccezione.
- Quello che succede è che non vengono semplicemente completate le restanti istruzioni del blocco a partire dal punto in cui è stata generata l'eccezione.

Exception Handler

- Naturalmente una volta generata l'eccezione occorre definire il punto da cui continuare l'esecuzione.
- Tale punto è detto *exception handler*.
- Esso segue subito il blocco try{..} che cattura l'eccezione.

Esempio

```
try {
   // Code that might generate exceptions
} catch(Type1 id1) {
   // Handle exceptions of Type1
} catch(Type2 id2) {
   // Handle exceptions of Type2
} catch(Type3 id3) {
   // Handle exceptions of Type3
}
```

La clausola catch

- Ogni clausola catch (exception handler) può essere vista come una metodo che ha un solo parametro: l'eccezione.
- Non si tratta di uno switch dove occorre la parola chiave break per evitare i diversi rami
- Un solo handler viene eseguito
- È sufficiente un solo handler anche se in più punti del blocco try viene generata la stessa eccezione

*Creare una propria eccezione

```
class SimpleException extends Exception {}

public class SimpleExceptionDemo {

public void f() throws SimpleException {
        System.out.println("Throw SimpleException from f()");
        throw new SimpleException()
;}

public static void main(String[] args) {
SimpleExceptionDemo sed = new SimpleExceptionDemo();
try {
        sed.f();
      } catch(SimpleException e) {System.err.println("Caught it!");}

}
```

La clausola throw delega la gestione dell'eccezione al chiamante

```
/**
 * Calcola la differenza in giorni fra due date, specificate rispettivamente dal giorno
 * "gg1", mese "mm1" e anno "aa1" e giorno "gg2", mese "mm2" e anno "aa2"
*/
public int differenzaDate(int gg1, int mm1, int aa1, int gg2, int mm2, int aa2)
 throws DataNonValida
 {
    if(!dataValida(gg1, mm1, aa1) || !dataValida(gg2, mm2, aa2))
        throw new DataNonValida();
    else {
        int risultato;
        // ... calcola la differenza fra le date
        return risultato;
    }
}
```

```
public void faiQualcosa() {
    boolean successo = false;
    int g1, g2, m1, m2, a1, a2;
    while (!successo) {
        // chiede all'utente di inserire valori per g1, m1, a1, g2, m2, a2
        try {
            ...
            int dd = differenzaDate(g1, m1, a1, g2, m2, a2);
            successo = true;
            ...
            System.out.println("La differenza è " + dd);
        } catch (DataNonValida dnv) {
            System.out.println("Almeno una delle date inserite non è valida");
        }
    }
}
```

```
public void faiQualcosa(int g1, int m1, int a1, int g2, int m2, int a2)
 throws DataNonValida
 {
    int dd = differenzaDate(g1, m1, a1, g2, m2, a2);
    System.out.println("La differenza è " + dd);
}
```

System.err

- System.err è lo standard di uscita per gli errori
- Corrisponde a System.out (video)
- L'utilità di avere due stream diversi sta nella possibilità di poter redirigire l'output conservando invariato l'errore

*Costruttore con stringa

```
class MyException extends Exception {
public MyException() {}
public MyException(String msg) {
   super(msg); }
}
```

Lo StackTrace

```
try {
   f();
    } catch(MyException e) {
        e.printStackTrace();}
```

Il metodo *printStackTrace()* stampa lo stack delle chiamate a procedura a partire dalla classe più esterna fino a quella in cui è stata generata l'eccezione.

Fornisce per ogni classe il numero di riga.

Molto più semplice il debug.

Altri metodi

- **String getMessage( )**
- **String getLocalizedMessage( )**

- **void printStackTrace( )**
- **void printStackTrace(PrintStream)**
- **void printStackTrace(java.io.PrintWriter)**

*Catturare qualunque eccezione

- Tutte le eccezioni ereditano la classe Exception
- Utilizzare un handler che cattura Exception equivale a catturare tutte le possibili ecezzioni.

```
try{ ... } catch(Exception e){
   e.printStackTrace(); }
```

Eccezioni generate automaticamente

- In alcuni casi la JVM genera automaticamente una eccezione:
  - Quando si usa un riferimento null *(**NullPointerException**)*
  - Quando si accede ad un elemento di un array oltre la sua lunghezza *(**ArrayOutOfBoundException**)*

- *Si pensi a quanto sarebbe oneroso controllare ogni volta che si usa un riferimento se questo è null*

Finally

- Alcune volte occorre effettuare delle operazioni in entrambe le eventualità di eccezione verificatasi o non.
- Il blocco finally viene eseguito sempre anche in caso di eccezione:

```
try{ …
   }catch(Exception e){

   }finally{ …}
```

Esempio-nofinally

```
public class OnOffSwitch {
private static Switch sw = new Switch();
public static void f() throws OnOffException1,OnOffException2 {}
public static void main(String[] args) {
try {
   sw.on();
   // Code that can throw exceptions...
   f();
   sw.off();
} catch(OnOffException1 e) {
    System.err.println("OnOffException1");
    sw.off();
} catch(OnOffException2 e) {
    System.err.println("OnOffException2");
     sw.off();
}
}
```

Esempio-finally

```
public class WithFinally {
static Switch sw = new Switch();
public static void main(String[] args) {
try {
   sw.on();
   // Code that can throw exceptions...
   OnOffSwitch.f();
} catch(OnOffException1 e) {
   System.err.println("OnOffException1");
} catch(OnOffException2 e) {
   System.err.println("OnOffException2");
} finally {
   sw.off();
}
}
}
```