Java

gestire gli errori con le eccezioni

G. Prencipe
prencipe@di.unipi.it

Introduzione

- La filosofia generale di Java è che un programma con errori non deve girare
- Una grossa parte viene fatta (come avete avuto modo di apprezzare) a tempo di compilazione
- Ci sono però errori che non possono essere rilevati in compilazione, ma solo a run-time

Errori

- C e svariati altri linguaggi prevedono il ritorno di particolari valori in caso di errore
- Tipicamente però il programmatore ignora questi valori, e quindi gli errori non vegono gestiti in maniera corretta
- Inoltre, controllare *tutti* questi valori produce tipicamente codice poco leggibile

Errori

- La soluzione è quella di forzare un formalismo per la cattura degli errori con la *gestione delle eccezioni*
  - Quando qualcosa va storto viene lanciata una eccezione
- Uno dei vantaggi con le eccezioni è che vengono *gestite da un gestore delle eccezioni*
  - Viene cioè introdotta separazione tra il codice eseguito quando tutto va bene e il codice eseguito quando qualcosa va male e viene generata una eccezione

Eccezioni basilari

- Una *condizione d'eccezione* è un problema che previene la continuazione del metodo in cui ci si trova
- Va distinta da un *problema normale* in cui si ha sufficiente informazione per risolvere la difficoltà
- Con una *condizione d'eccezione* non si hanno sufficienti informazioni (nel contesto in cui avviene) per proseguire, e si delega la risoluzione del problema a un contesto più ampio

Errori

- Un tipico esempio è la divisione per zero
- Se nel contesto in cui siamo abbiamo sufficienti informazioni per poter controllare il valore del denominatore, tutto va bene
- Nel caso in cui il denominatore a zero è un valore inatteso, si *lancia* una eccezione piuttosto che continuare nel normale flusso d'esecuzione

Lanciare eccezioni

- Quando si lancia una eccezione, accadono diverse cose
  - Viene creato un *oggetto eccezione*
    - Sull'heap con una new
  - Il flusso corrente d'esecuzione viene bloccato e il riferimento per l'oggetto eccezione viene restituito dal contesto corrente
  - Il meccanismo di gestione dell'eccezione entra in gioco, e cerca un posto opportuno per continuare l'esecuzione del programma
    - Questo posto è il gestore delle eccezioni , il cui compito è di cercare di risolvere il problema

Lanciare eccezioni

- Come semplice esempio, si consideri un riferimento a un oggetto **t**
- Prima di utilizzarlo, vogliamo controllare che sia stato inizializzato, altrimenti viene creata una informazione (eccezione) *lanciata* in un contesto più ampio

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

- In questo modo la risoluzione del problema viene delegata

Argomenti delle eccezioni

- Come per tutti gli oggetti Java, anche per le eccezioni vengono invocati costruttori
- Ci sono due costruttori in tutte le eccezioni standard
  - Il primo: costruttore di **default**
  - Il secondo: prende una **String**a come argomento così da poter inserire nella eccezione informazioni pertinenti

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

  - Vedremo come poter estrarre questa stringa

Argomenti eccezioni

- È possibile lanciare qualsiasi oggetto di tipo **Throwable** (la classe radice delle eccezioni)
- Tipicamente viene lanciata una classe differente di eccezioni per ogni diverso tipo di errore
- L'informazione relativa all'errore è rappresentata sia dentro l'oggetto eccezione che implicitamente nel nome della classe dell'eccezione
  - Tipicamente l'unica informazione è legata al tipo dell'eccezione

Catturare eccezioni

- Se un metodo lancia un'eccezione, si assume che l'eccezione verrà *catturata*
- Per capire come viene catturata un'eccezione, bisogna introdurre le *guarded regions*
  - Sezioni di codice che potrebbero produrre una eccezione, seguite dal codice che le gestisce

Il blocco **try**

- Se siamo in un metodo e viene lanciata un'eccezione, il metodo termina nel processo di *lancio*
- Se non vogliamo che **throw** termini il metodo, si deve scrivere uno speciale blocco di codice all'interno del quale *catturare* l'eccezione
  - Il blocco **try**

Il blocco **try**

- Il blocco **try** è un normale blocco di codice preceduto dalla parola chiave **try**

```
try {
   // Codice che potrebbe generare l'eccezione
}
```

- In questo modo è possibile inserire tutto il codice da controllare in un unico blocco e catturare le eccezioni in un solo posto

Gestori delle eccezioni

- I gestori delle eccezioni seguono immediatamente il blocco **try** e sono denotati dalla parola chiave **catch**

```
try {
 // Codice che potrebbe generare l'eccezione
 } catch (Tipo1 id1) {
      // Gestisce l'eccezione di Tipo1
 } catch (Tipo2 id2) {
      // Gestisce l'eccezione di Tipo1
 }
```

Gestori delle eccezioni

- Ogni catch (gestore dell'eccezione) prende uno ed un solo argomento di un tipo particolare
  - L'identificatore per quel tipo (**id1**, **id2**) può essere utilizzato all'interno del gestore come un normale argomento di un metodo
- I gestori devono essere inseriti subito dopo il blocco **try**
- Se viene lanciata un'eccezione, il meccanismo di gestione delle eccezioni cerca il primo gestore il cui argomento è compatibile con il tipo dell'eccezione lanciata, e entra nella **catch** corrispondente
  - La ricerca termina dopo l'esecuzione della **catch**

Creare eccezioni

- In piena filosofia Java, è possibile creare le proprie eccezioni
- Per fare questo bisogna ereditare da una classe d'eccezioni già esistente
  - Preferibilmente da una che è vicina al tipo dell'eccezione che vogliamo introdurre
- Il modo più semplice è di lasciar creare al compilatore il costruttore di default

Esempio

```
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) {
      // Il messaggio d'errore è inviato alla console per lo standard error
      System.err.println("Caught it!"); }
  }
} ///:~
```

Creare eccezioni

- Nell'esempio visto si utilizza solo il costruttore di default
- È possibile definire anche il costruttore che prende come argomento **String**, aggiungendo qualche riga di codice in più

Esempio

```
class MyException extends Exception {
  public MyException() {}
  public MyException(String msg) { super(msg); }
}
public class FullConstructors {
    public static void f() throws MyException {
          System.out.println("Throwing MyException from f()");
          throw new MyException();
    }
    public static void g() throws MyException {
          System.out.println("Throwing MyException from g()");
          throw new MyException("Originated in g()");
    }
/* continua */
```

Esempio

```
public static void main(String[] args) {
    try {
      f();
    } catch(MyException e) {
     // printStacktrace è un metodo della classe Throwable
     ///(superclasse di Exception)
     // Produce informazioni sulla sequenza di metodi invocati
     // per arrivare al punto in cui è avvenuta l'eccezione
      e.printStackTrace(); }
    try {
      g();
    } catch(MyException e) {
      e.printStackTrace();
    }
  }} ///:~
```

Creare eccezioni

- È possibile spingersi oltre, e creare ulteriori costruttori e membri
- Vediamo un esempio: **ExtraFeatures.java**

Specifica delle eccezioni

- In Java si incoraggia a informare chi utilizzerà un certo metodo delle possibili eccezioni che potrebbero essere generate
- Chiaramente, se il cliente di un certo metodo ha il codice del metodo, potrebbe cercare i **throw** presenti nel codice e capire da essi quali eccezioni potrebbero essere generate
- Questo non è sempre possibile, e quindi Java fornisce una particolare sintassi per poter *specificare le eccezioni* che un certo metodo può lanciare

Specifica delle eccezioni

- Si utilizza la parola chiave **throws**, seguita dalla lista delle possibili eccezioni

  void f() throws A, B, C {….}
- Se il codice di un metodo genera eccezioni che non vengono catturate, il compilatore si lamenta. In questo caso bisogna
  - *Catturare* l'eccezione, o
  - Indicare una *specifica delle eccezioni*

Specifica delle eccezioni

- È possibile dichiarare una specifica per delle eccezioni che poi non vengono generate nel corpo del metodo
- Questo comunque forza chi utilizza il metodo a gestire quella eccezione
  - Questo è utile se in una prima stesura del metodo le eccezioni non vengono generate, ma in successive modifiche si (il cliente non deve cambiare nulla, avendo già previsto la gestione delle eccezioni)

Catturare qualsiasi eccezione

- È possibile definire un gestore che catturi qualsiasi eccezione
- Per fare questo si deve catturare l'eccezione della superclasse Exception

```
catch(Exception e) {
  System.out.println("Catturata una eccezione");
}
```

- È preferibile inserire una **catch** di questo tipo dopo tutte le altre **catch** più specifiche, per evitare che esse vengano ignorate

Throwable

- La classe Exception è la classe base di tutte le eccezioni, quindi non si ottengono particolari informazioni sul tipo dell'eccezione generata
- È possibile comunque invocare tutti i metodi di **Throwable** (da cui eredita **Exception**)

Throwable

- String getMessage
  - Restituisce il messaggio legato all'eccezione (*detail message*)
- String toString()
  - Restituisce una breve descrizione del **Throwable**, includendo il *detail message*
- void printStackTrace()
  - Stampa la sequenza di metodi invocati fino al punto in cui è avvenuta l'eccezione (stampa lo *stack trace* dell'eccezione)

Rilanciare un'eccezione

- A volte si vuole rilanciare un'eccezione appena catturata

```
catch (Exception e) {
   System.err.println("Eccezione Lanciata");
   throw e;
}
```

- In questo caso l'eccezione viene rilanciata al gestore delle eccezioni nel contesto più ampio
- Ogni successiva clausola **catch** nello stesso blocco **try** viene ignorata

Exception chaining

- È possibile lanciare un'eccezione mentre se ne gestisce un'altra, cioè nel corpo di una **catch**
- Inoltre è possibile passare al costruttore di una eccezione come parametro un'altra eccezione

Esempio

```
….
catch(NullPointerException e) {
   ….
   throw new ClassCastException(e)
}
```

Eccezioni standard di Java

- La classe **Throwable** descrive qualsiasi cosa che può essere lanciata come eccezione
- Ci sono due tipi (sottoclassi) generali per **Throwable**
  - **Error**: rappresenta errori a tempo di compilazione e errori di sistema che non ci preoccupiamo di catturare
  - **Exception**: può essere lanciata da una qualsiasi dei metodi delle librerie standard Java e dai metodi scritti da noi e da errori a run-time
  - La cosa migliore per avere un'idea delle eccezioni standard disponibili è di scorrere la documentazione
    - Proviamo a scorrere la documentazione per **Throwable** e **Exception**

Eccezioni standard di Java

- L'idea di base delle eccezioni è che il loro nome rappresenta il problema che è accaduto
- Le eccezioni non sono tutte definite in **java.lang**
  - Ne esistono anche in **util**, **net** e **io**

Il caso **RunTimeException**

- Ci sono una serie di eccezioni che Java lancia automaticamente
- Consideriamo l'esempio iniziale

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

- Sarebbe oneroso dover controllare ogni riferimento per verificare che non sia **null**
  - Fortunatamente questa è una cosa che Java fa automaticamente, e se viene effettuata una chiamata a un riferimento **null**, viene generata la **NullPointerException** automaticamente

Il caso **RunTimeException**

- C'è un gruppo di ecccezioni che rientra in questa categoria
  - Esse sono automaticamente lanciate da Java
- Sono raggruppate in una singola classe chiamata RuntimeException
  - Rappresenta un perfetto esempio di ereditarietà: stabilisce una famiglia di tipi che hanno alcune caratteristiche e comportamenti in comune

Il caso **RunTimeException**

- Inoltre non è necessario inserire una specifica d'eccezione nei metodi con una **throw** di una **RuntimeException**
  - Tutto automatico
- Non è nemmeno necessario fare una **catch** di queste eccezioni
  - Esse indicano la presenza di bachi nel codice
  - Se le catturassimo, sapremmo del baco e lo potremmo correggere!!

Il caso **RunTimeException**

- Dato che il compilatore non obbliga la specifica di queste eccezioni, è plausibile che una **RuntimeException** si propaghi fino al **main** senza essere catturata
- In questo caso, viene invocata automaticamente la **printStackTrace()**, e viene stampato a console lo *stack trace* dell'eccezione
  - Viene stampata cioè la lista dei metodi coinvolti dal punto in cui è avvenuta l'eccezione fino al **main**

Il caso **RunTimeException**

- Queste sono le *uniche* eccezioni che possono essere ignorate
  - Il compilatore obbliga la gestione di tutte le altre
- Questo perché esse rappresentano errori di programmazione
  - Accesso a un oggetto tramite riferimento **null**
  - Accesso oltre i limiti di un array
    - **ArrayIndexOutOfBoundsException**

La clausola **finally**

- Ci possono essere dei pezzi di codice che vogliamo eseguire comunque, indipendentemente dal fatto che una eccezione sia stata lanciata in un blocco **try**
- Per ottenere questo effetto si ricorre alla clausola **finally** alla fine dei gestori delle eccezioni

La clausola **finally**

- Quindi, la struttura completa di un blocco **try** è

```
try {
   // Codice che potrebbe lanciare eccezioni A o B
} catch (A a1) {
   // Gestore per l'eccezione A
} catch (B b1) {
   // Gestore per l'eccezione A
} finally {
   // Attività da eseguire sempre
}
```

Esempio -- la clausola **finally**

- Per dimostrare che la clausola finally viene sempre eseguita, vediamo un esempio: **FinallyWorks.java**
- Si note come la clausola **finally** venga *sempre* eseguita
- L'esempio mostrato ci fornisce anche una tecnica riprovare a eseguire un pezzo di codice dopo che si è verificata una eccezione
  - In genere, infatti, dopo la gestione dell'eccezione si esegue il codice seguente
  - Inserendo il blocco **try** in un ciclo si riprova a eseguire il codice, cercando di sistemare (nella **catch**) la causa dell'eccezione

Perché **finally**?

- **finally** si rende tipicamente necessaria quando si ha la necessità di riportare qualcosa che *non sia* la memoria in un certo stato
  - Ricordiamo che della memoria si occupa il *garbage collector*
  - Ad esempio: chiudere un file, chiudere una connessione di rete, cancellare qualcosa disegnata sullo schermo, ecc.

La clausola **finally**

- Anche se l'eccezione generata non è catturata (da una **catch**) nel contesto corrente, la clausola **finally** viene eseguita prima che il meccanismo di gestione dell'eccezione continui la ricerca per un gestore nel contesto più ampio
- Vediamo un esempio: **AlwaysFinally.java**

Restrizioni

- Quando si fa *overriding* di un metodo, si possono lanciare solo le eccezioni che sono state specificate nella versione del metodo presente nella superclasse
- Questa restrizione è utile: in questo modo, infatti, il codice che funziona per la superclasse funzionerà automaticamente per ogni oggetto derivato da essa

Restrizioni

- Questa restrizione non si applica ai costruttori delle sottoclassi, che possono lanciare le eccezioni che vogliono
- L'unica nota è che, dato che il costruttore di una sottoclasse automaticamente invoca il costruttore della superclasse, tutte le eccezioni presenti nella superclasse devono essere specificate anche nel costruttore della sottoclasse
  - Se ne possono aggiungere altre, a differenza dei metodi normali con *overriding*

Costruttori

- Bisogna porre attenzione nella gestione delle eccezioni nei costruttori
- Come sappiamo, il costruttore pone l'oggetto in uno stato sicuro
- Il lancio di eccezioni nel costruttore potrebbe rendere le cose inconsistenti
  - In questo caso (eccezione generata) vorremmo effettuare delle operazioni per sistemare le cose
  - Queste operazioni non le vogliamo però eseguire sempre (solo se lo stato diviene inconsistente)
  - Potremmo utilizzare **finally** in combinazione con un *flag* che ci dice se lo stato è ok o meno, ma non è elegante

Costruttori

- Per chiarire le cose, vediamo un esempio in cui si gestiscono file: **Cleanup.java**
  - Viene utilizzata la classe **BufferedReader** e **FileReader** dalla libreria di I/O di Java:

Match delle eccezioni

- Come già accennato, quando viene lanciata un'eccezione, il sistema di gestione delle eccezioni cerca il gestore "più vicino" (nell'ordine in cui sono scritti) che ha un *match* con il tipo dell'eccezione generata
- Il *match* tra il tipo dell'eccezione e il suo gestore non deve essere perfetto
  - Un gestore scritto per catturare una eccezione di un certo tipo **E**, catturerà ovviamente eccezioni di tipo **E**, ma anche eccezioni derivate da E

Esempio -- match delle eccezioni

```
class Annoyance extends Exception {}
class Sneeze extends Annoyance {}
public class Human {
public static void main(String[] args) {
   try {
     throw new Sneeze();
   } catch(Annoyance a) {
   // L'eccezione Sneeze() viene catturata qui
   // dato che Sneeze è derivato da Annoyance
     System.err.println("Caught Annoyance");
   }
 }} ///:~
```

Match delle eccezioni

- In questo modo, se si decide di introdurre eccezioni derivate da una certa eccezione base, non bisogna cambiare nulla nel codice già scritto, a patto che veniva già catturata l'eccezione base
- Nota: se si cerca di "mascherare" l'eccezione derivata scrivendo prima le clausole **catch** della eccezione base e poi quelle per l'eccezione derivata, il compilatore si lamenta dicendo che l'eccezione derivata non può essere raggiunta

Match delle eccezioni

- Cioè, questo *non* va bene:

```
try {
    throw new Sneeze();
  } catch(Annoyance a) {
    System.err.println("Caught Annoyance");
  } catch(Sneeze s) {
    System.err.println("Caught Sneeze);
  }
```

Esercizi

1. Creare una classe con un **main** che lancia un oggetto della classe **Exception** in un blocco **try**
   1. Passare al costruttore per **Exception** un argomento **String**a
   2. Catturare l'eccezione nella clausola **catch** e stampare l'argomento **String**a
   3. Aggiungere la clausola **finally** che stampa un messaggio

Esercizi

2. In un file **Eccezione.java**, creare una nuova eccezione **NuovaEccezione** utilizzando **extends**
   1. Scrivere un costruttore per questa classe che prende come argomento una **String**a e la memorizza nell'oggetto con un riferimento a **String**a
   2. Aggiungere un metodo che stampi la **String**a memorizzata
   3. Creare una clausola **try-catch** per testare la nuova classe
   4. Scrivere una classe con un metodo che lancia una **NuovaEccezione** e testare il codice scritto nel main

Esercizi

3. Scrivere un pezzo di codice che genera e cattura una **ArrayIndexOutOfBoundsexception**
4. Creare in **Eccezione2.java** 3 nuovi tipi di eccezioni
   1. Scrivere una classe con un metodo che le lanci tutte e tre
   2. Nel **main**, invocare il metodo utilizzando una sola **catch** per catturare tutte e tre le eccezioni

Esercizi

5. Creare una gerarchia a 3 livelli di nuove eccezioni in **Eccezione3.java**
   1. Creare una classa **A** con un metodo che lancia un'eccezione alla base della gerarchia
   2. Ereditare **B** da **A** e fare *overriding* del metodo in modo che lanci un eccezione al secondo livello della gerarchia
   3. Ripetere ereditando **C** da **B**
   4. Nel **main** creare **C**, fare upcast ad **A**, e invocare il metodo

Java

gestire gli errori con le eccezioni

fine