ECCEZIONI IN JAVA

1

Generazione di "errori"

- Un metodo puo' richiedere che gli argomenti attuali soddisfino determinate precondizioni per procedere nell'esecuzione
  - mthd(List L) con L non vuota
- Componenti esterni potrebbero fallire
  - File non presente
- Implementazioni parziali
  - Modulo con alcune funzionalita' non ancora implementate
- Come gestiamo queste situazioni "anomale"?

2

Gestione errori

- Diverse tecniche
  - Parser per gli errori sintattici
  - Tecniche di analisi statica (type checker) per gli errori semantici
  - Test covering & Best practices
  - Ignorare gli errori
- Ora noi vediamo il meccanismo delle eccezioni: meccanismi linguistici che permettono di traferire il controllo del programma dal punto in cui viene rilevato l'errore al codice che permette di gestirlo

3

Cosa sono?

- Le eccezioni sono dei particolari oggetti usati per rappresentare e catturare condizioni anomale del comportamento di programmi
  - Esempio comportamenti anomali durante le operazioni di I/O, null pointer, …
- *Sollevare* (*throwing*) una eccezione significa attivare una sorta di uscita di emergenza nell'esecuzione del programma
- *Catturare* (*catching*) una eccezione significa programmare le azioni da eseguire per gestire il comportamento anomalo

4

Perche' sono utili?

- Compilatore non e' in grado di determinare tutti gli errori
- Separation of concerns: separare il codice di gestione degli errori dal codice "normale"
  - Chiarezza del codice (debugging)
  - Raggruppare e differenziare i tipi delle situazioni di comportamento anomalo che si possono presentare

5

Esempio

```
public class ArrayExceptionExample {
  public static void main( String args[ ]) {
      String [ ] colori = {"Rossi", "Bianchi", "Verdi"};
      System.out.println(colori[3]);
  }
}
```

Cosa succede quando compiliamo e poi mandiamo il programma in esecuzione?

6

Esempio

```
public class ArrayExceptionExample {
  public static void main( String args[ ]) {
      String [ ] colori = {"Rossi", "Bianchi", "Verdi"};
      System.out.println(colori[3]);
  }
}
```

Compilazione OK ma a runtime ...

ArrayExceptionExampleException in thread "main"
java.lang.ArrayIndexOutOfBoundsException:
3 at ArrayExceptionExample.main(ArrayExceptionExample.java:6)

Formato dei messaggi

[exception class]:
[additional description of exception] at
[class].[method]([file]: [line number])

Formato

- java.lang.ArrayIndexOutOfBoundsException: 3 at ArrayExceptionExample.main(ArrayExceptionExample.java:6)
- **Exception Class**
  - java.lang.ArrayIndexOutOfBoundsException
- **Quale indice dell'array (additional information)**
  - 3
- **Quale metodo solleva l'eccezione**
  - ArrayExceptionExample.main
- **Quale file contiene il metodo**
  - ArrayExceptionExample.java
- **Quale linea del file solleva l'eccezione**
  - 6

Eccezioni a runtime

- Abbiamo visto la situazione in cui le situazioni anomale provocano a run-time la terminazione (anomala) del programma in esecuzione.
- Domanda: e' possibile prevedere meccanisni linguistici che permettono di affrontare le situazioni anomale come un "normale" problema di programmazione?

Codificare le anomalie

- Prevedere opportuni meccanismi di codifica per le situazioni anomale
  - ArrayOutOfBound: l'accesso all'array fuori dalle dimensione restituisce il valore "-1" che codifica l'anomalia
  - L'accesso a un file non presente nello spazio del programma restituisce la stringa "null"
  - E' fattibile? Tecnica scalabile?
- Il modo moderno di affrontare questo aspetto e' quello di introdurre specifici meccanismi linguistici
  - Ocaml (failwith), Java (throw+try-catch), C++, C# .....

Java: Sollevare eccezioni

- Il linguaggio prevede una primitiva specifica per dichiarare e programmare il modo in cui le eccezioni sono sollevate
- Usare il costrutto **throw** all'interno del codice dei metodi

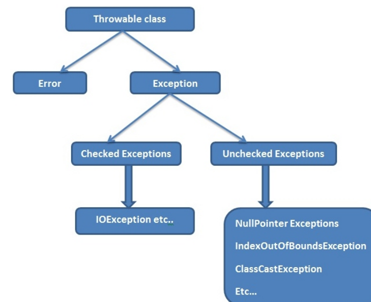
```
if (myObj == null)
    throw new NullPointerException()
```

throw

- Il costrutto **throw** richiede come argomento un oggetto che abbia come tipo un qualunque sottotipo di **Throwable**
- La classe **Throwable** contiene tutti i tipi di errori e di eccezioni
- Come si fa a vedere la struttura?
  - Consultate la documentazione on line delle API
  - http://docs.oracle.com/javase/7/docs/api/java/lang/Throwable.html

13

Java Exception Hierachy (concettuale)



14

Dichiarare eccezioni

- Se un metodo contiene del codice che può generare una eccezione allora si può dichiarare nella dichiarazione del metodo questa possibilità. Esempi:
  - ✓ `public void myMethod() throws DivideByZeroException, ArrayIndexOutOfBoundException {…}`
  - ✓ `public void myMethod() throws IOException {…}`
- L'eccezione diventa pertanto una componente del tipo del metodo

15

Gestione delle eccezioni

- Java prevede strumenti linguistici per programmare la gestione delle eccezioni
- Clausola

```
try {
   // codice che puo' sollevare l'eccezione
}
catch ([tipo eccezione] e) {
   // codice di gestione della eccezione
}
```

16

Gestioni multiple

- E' possibile programmare una gestione "multipla" delle eccezioni

```
try {
  // codice che può sollevare diverse eccezioni
}
catch (IOException e) {
  // Gestione IOException
}
catch (ClassNotFoundException e) {
  // Gestione ClassNotFoundException
}
```

- Le clausole **catch** vengono analizzate in sequenza, e si eseguono i comandi della prima alla cui classe l'eccezione sollevata appartiene.

17

La clausola *finally*

- La clausola *finally* permette di programmare del codice di clean-up indipendentemente da quello che e' successo nel codice monitorato

```
try {
  // codice che puo' sollevare diverse eccezioni
}
catch ([Tipo Eccezione] e) {
  // gestione Exception
}
finally {
  // Codice di clean up che viene sempre eseguito
}
```

- Viene usata principalmente per rialsciare risorse utilizzate dal metodo (esempio: chiudere un file)

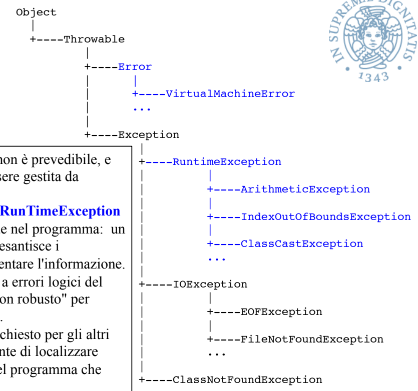
18

Eccezioni *checked* e *unchecked*

- Le eccezioni si dividono in due categorie:
  - Eccezioni **controllate** (**checked**)
  - Eccezioni **non controllate** (**unchecked**)
- Le eccezioni ***controllate*** DEVONO essere gestite esplicitamente dal programma, altrimenti il compilatore segnalerà un errore. Ogni volta che scriviamo un'istruzione che potrebbe lanciare un'eccezione ***controllata***, allora
  - l'istruzione deve essere racchiusa in un blocco **try-catch** che possa gestire quel tipo di eccezione

  oppure
  - il metodo che contiene l'istruzione deve ***delegare*** la gestione dell'eccezione al chiamante, con la clausola **throws**

Eccezioni checked e unchecked



- Una istanza di **Error** non è prevedibile, e comunque non può essere gestita da programma.
- Una eccezione di tipo **RunTimeException** può verificarsi ovunque nel programma: un controllo esplicito appesantisce i programmi senza aumentare l'informazione.
- Corrispondono spesso a errori logici del programma (codice "non robusto" per mancanza di controlli).
- Il controllo esplicito richiesto per gli altri tipi di eccezioni consente di localizzare rapidamente le parti del programma che potrebbero lanciarle.

Il nostro esempio

```
public class ArrayExceptionExample {
  public static void main( String args[ ]) {
      String [ ] colori = {"Rossi", "Bianchi", "Verdi"};
      System.out.println(colori[3]);
  }
}
```

ArrayExceptionExampleException in thread "main"
java.lang.ArrayIndexOutOfBoundsException:
3 at ArrayExceptionExample.main(ArrayExceptionExample.java:6)

Esempio di una eccezione unchecked (runtime)
Eccezioni unchecked: il metodo non deve necessariamente prevedere il codice di gestione

Ricapitoliamo

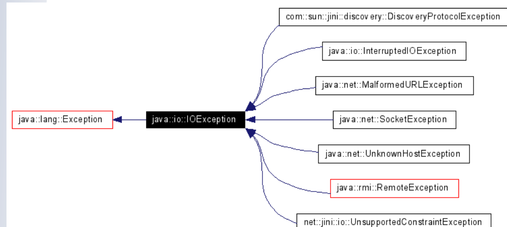
- i tipi di eccezione sono particolari classi di Java che
  - contengono solo il costruttore
    - ✓ci possono essere più costruttori overloaded
  - sono definite in "moduli" separati da quelli che contengono i metodi che le possono sollevare
- le eccezioni sono oggetti
  - creati eseguendo new di un exception type e quindi eseguendo il relativo costruttore
- ✓ esiste una gerarchia "predefinita" di tipi relativi alle eccezioni
  - – nuovi tipi di eccezioni possono essere definiti dall'utente e sono collocati nella gerarchia con l'usuale extends

Esempi

java.lang
Class NullPointerException

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.lang.RuntimeException
        java.lang.NullPointerException

esempi



Definire tipi di eccezione

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

- è checked (perché non estende **RuntimeException**)
- definisce solo un costruttore
  - invocato quando si crea una istanza con la new
  - il costruttore può avere parametri
- il corpo del costruttore riutilizza semplicemente il costruttore del supertipo
  - perché deve passargli il parametro?
- una new di questa classe provoca la creazione di un nuovo oggetto che "contiene" la stringa passata come parametro

25

Costruire oggetti eccezione

public class NuovoTipoDiEcc extends Exception {
 public NuovoTipoDiEcc(string s) {super(s);}
}

- una new di questa classe provoca la creazione di un nuovo oggetto che "contiene" la stringa passata come parametro

Exception e = new NuovoTipoDiEcc ("Questa è la causa") ;
String s = e.toString() ;

- la variabile s punta alla stringa "NuovoTipoDiEcc: Questa è la causa"

26

Sollevare eccezioni

- una procedura può terminare
  - (ritorno normale) con un return
  - (ritorno di una eccezione) con un throw

```
public static int fact (int n) throws
  NonpositiveExc
  // EFFECTS: se n>0, ritorna n!
  // altrimenti solleva NonpositiveExc
{ if (n <= 0) throw new
  NonPositiveExc("Num.fact");
...}
```

- la stringa contenuta nell'eccezione è utile soprattutto quando il programma non è in grado di "gestire" l'eccezione
  - permette all'utente di identificare la procedura che la ha sollevata
  - può comparire nel messaggio di errore che si stampa subito prima di forzare la terminazione dell'esecuzione

27

Gestire eccezioni

- quando una procedura termina con un throw
  - l'esecuzione non riprende con il codice che segue la chiamata (call-return tradizionale)
  - il controllo viene trasferito ad un pezzo di codice preposto alla gestione dell'eccezione
- due possibilità per la gestione
  - gestione esplicita quando l'eccezione è sollevata all'interno di uno statement try
    - ✓ in generale, quando si ritiene di poter recuperare uno stato consistente e di portare a termine una esecuzione quasi "normale"
  - gestione di default, mediante propagazione dell'eccezione alla procedura chiamante
    - ✓ possibile solo per eccezioni non checked o per eccezioni checked elencate nell'header della procedura che riceve l'eccezione

28

Gestione esplicita delle eccezioni

- gestione esplicita: l'eccezione è sollevata all'interno di uno statement try
- codice per gestire l'eccezione NonPositiveExc eventualmente sollevata da una chiamata di fact

try { x = Num.fact (y); }
catch (NonPositiveExc e) {
 // qui possiamo usare e, cioè l'oggetto eccezione
}

- la clausola catch non deve necessariamente identificare il tipo preciso dell'eccezione, ma basta un suo supertipo

try { x = Arrays.searchSorted (v, y); }
catch (Exception e) { s.Println(e); return;}
 // s è una PrintWriter

- segnala l'informazione sia su **NullPointerExc** che su **NotFoundExc**

29

Try e Catch annidati

try { ...;
 try { x = Arrays.searchSorted (v, y); }
 catch (NullPointerExc e) {
 throw new NotFoundExc ();}
 }
catch (NotFoundExc b) {...}

- la clausola catch nel try più esterno cattura l'eccezione NotFoundExc se è sollevata da searchSorted o dalla clausola catch più interna

30

Catturare eccezioni unchecked

- le eccezioni unchecked sono difficili da catturare: una qualunque chiamata di procedura può sollevarle difficile sapere da dove vengono

```
try { x = y[n]; i = Arrays.searchSorted (v, x); }
catch (IndexOutOfBoundsException e) {
  // cerchiamo di gestire l'eccezione pensando che sia
  // stata sollevata da x = y[n]
}
// continuiamo supponendo di aver risolto il problema
```

- ma l'eccezione poteva venire dalla chiamata a searchSorted
- l'unico modo per sapere con certezza da dove viene è restringere lo scope del comando try

31

Aspetti metodologici

- gestione delle eccezioni
  - riflessione
  - mascheramento
- quando usare le eccezioni
- come scegliere tra checked e unchecked
- defensive programming

32

Gestione delle eccezioni

- se un metodo chiamato da p ritorna sollevando una eccezione, anche p termina sollevando un'eccezione
  - usando la propagazione automatica
    - ✓della stessa eccezione (NullPointerException)
  - catturando l'eccezione e sollevandone un'altra
    - ✓possibilmente diversa (EmptyException)

33

Gestione delle eccezioni

```
public static int min (int[] a) throws NullPointerException,
                                        EmptyException
// se a è null solleva NullPointerException
// se a è vuoto solleva EmptyException
// altrimenti ritorna il minimo valore in a

{int m;
  try { m = a[0]}
      catch(IndexOutOfBoundsException e) {
       throw new EmptyException("Arrays.min");
      }
for (int i = 1; i < a.length ; i++)
  if (a[i] < m) m = a[i];
return m;
}
```

Nota: usiamo le eccezioni (catturate) al posto di un test per verificare se a è vuoto

34

Gestione delle eccezioni via mascheramento

- se un metodo chiamato da p ritorna sollevando una eccezione, p gestisce l'eccezione e ritorna in modo normale

```
public static boolean sorted (int[] a) throws
    NullPointerException
// se a è null solleva NullPointerException
// se a è ordinato in senso crescente ritorna true
// altrimenti ritorna false
{int prec;
  try { prec = a[0]}
       catch(IndexOutOfBoundsException e) { return true;}
for (int i = 1; i < a.length ; i++)
  if (prec <= a[i]) prec = a[i]; else return false;
return true;
}
```

35

Quando usare le eccezioni

- le eccezioni non sono necessariamente errori
  - ma metodi per richiamare l'attenzione del chiamante su situazioni particolari (classificate dal progettista come eccezionali)
- comportamenti che sono errori ad un certo livello, possono non esserlo affatto a livelli di astrazione superiore
  - IndexOutOfBoundsException segnala chiaramente un errore all'interno dell'espressione a[0] ma non necessariamente per le procedure min e sort
- il compito primario delle eccezioni è di ridurre al minimo i vincoli della specifica per evitare di codificare informazione su terminazioni particolari nel normale risultato

36

Checked o unchecked

- le eccezioni checked offrono una maggiore protezione dagli errori
  - sono più facili da catturare
  - il compilatore controlla che l'utente le gestisca esplicitamente o per lo meno le elenchi nell'header, prevedendone una possibile propagazione automatica
    - ✓se non è così, viene segnalato un errore
- le eccezioni checked sono pesanti da gestire in quelle situazioni in cui siamo ragionevolmente sicuri che l'eccezione non verrà sollevata
  - perché esiste un modo conveniente ed efficiente di evitarla o per il contesto di uso limitato
  - solo in questi casi si dovrebbe optare per una eccezione unchecked

Defensive programming

- l'uso delle eccezioni facilita uno stile di progettazione e programmazione che protegge rispetto agli errori
  - anche se non sempre un'eccezione segnala un errore
- fornisce una metodologia che permette di riportare situazioni di errore in modo ordinato
  - senza disperdere tale compito nel codice che implementa l'algoritmo
- nella programmazione *defensive*, si incoraggia il programmatore a verificare l'assenza di errori ogniqualvolta ciò sia possibile
  - ed a riportarli usando il meccanismo delle eccezioni
  - un caso importante legato alle implementazioni parziali

Metodi e eccezioni

- Con le eccezioni i metodi tendono a diventare totali
  - ma non è sempre possibile
- Chi invoca il metodo dovrebbe farsi carico di effettuare tale controllo
  - sollevando una eccezione
    - ✓questa eccezione può essere catturata, magari ad un livello superiore
    - ✓si suggerisce di usare in questi casi una eccezione generica unchecked FailureException

Checked vs unchecked

Pro Checked Exceptions:
Compiler enforced catching or propagation of checked exceptions make it harder to forget handling that exception.

Pro Checked Exceptions:
Unchecked exceptions makes it easier to forget handling errors since the compiler doesn't force the developer to catch or propagate exceptions (reverse of 1).

Pro Unchecked Exceptions:
Checked exceptions that are propagated up the call stack clutter the top level methods, because these methods need to declare throwing all exceptions thrown from methods they call.

Pro Checked Exceptions:
When methods do not declare what unchecked exceptions they may throw it becomes more difficult to handle them.

Pro Unchecked Exceptions:
Checked exceptions thrown become part of a methods interface and makes it harder to add or remove exceptions from the method in later versions of the class or interface.

Riferimenti

Anders Hejlsberg on checked vs. unchecked exceptions
http://www.artima.com/intv/handcuffs.html

James Gosling on checked exceptions
http://www.artima.com/intv/solid.html

Bill Venners on Exceptions
http://www.artima.com/interfacedesign/exceptions.html

Bruce Eckel on checked exceptions
http://www.artima.com/intv/typingP.html

Designing with Exceptions (Bill Venners - www.artima.com)
http://www.artima.com/designtechniques/desexcept.html

Effective Java (Joshua Bloch - Addison Wesley 2001)

Daniel Pietraru - in favor of checked exceptions
Exceptional Java - Checked exceptions are priceless… For everything else there is the RuntimeException