Programmazione a Oggetti

Eccezioni

Sommario

Eccezioni

Generalità, motivazioni

Eccezioni in Java

Sollevamento

Cattura

Trattamento

Eccezioni

I programmi spesso si trovano ad operare in condizioni anomale:

Il programma riceve un ***dato sbagliato***

Tenta di stabilire una connessione remota che fallisce (es: ***time-out****)*

Anche errori del ***run-time system****:*

- divisione per zero

- invocazione di un metodo su ***riferimento nullo***

- `(S)ref` *con tipo dinamico di* `ref` *non sottotipo di S*

- `new C()` *e C non trovata dal class loader*

Esempio: funzioni parziali

```
public static int mcd(int m, int n)
/* PREC: m,n > 0
 * POST: ritorna l'MCD tra m e n
 */
```

Come comportarsi su input negativi o nulli?

Soluzione 1: supporre che il chiamante rispetti la precondizione!

Soluzione 2

```
public static int mcd(int m, int n)
/* POST: se m, n > 0 ritorna l'MCD
  * tra m e n, altrimenti torna 0
  */
```

La funzione mcd viene resa arbitrariamente totale

La responsabilità di verificare la correttezza della chiamata a mcd ricade sul chiamante

Soluzione 3

```
public static int mcd(int m, int n)
/* POST: se m, n > 0 ritorna l'MCD
  * tra m e n, altrimenti torna 0
  */
```

Il programmatore si disinteressa dell'output dato dalla funzione

La responsabilità di verificare la correttezza della chiamata a mcd ricade sul chiamante

Problemi

Chiamare una funzione con argomenti illegali è probabilmente un errore (andrebbe opportunamente segnalato)

Il range di una funzione a volte è tutto l'insieme dei valori del tipo di ritorno

In ogni caso: utile un meccanismo di astrazione che permetta a una funzione o di tornare un valore o terminare in modo eccezionale

Soluzione 3

```
public static int mcd(int m, int n)
    throws NonPositiveException
/* POST: se m, n > 0 ritorna l'MCD
 * tra m e n, altrimenti solleva
 * l'eccezione NonPositiveException
 */
```

Dichiarare che la funzione può terminare in modo eccezionale, a causa di un errore che si può verificare durante l'esecuzione

Eccezioni: generalità

Eccezioni. ***Tre momenti****:*

Sollevamento *dell'eccezione (****trhowing exception****) viene generato una segnalazione di un evento anomalo*

Cattura *dell'eccezione (****catching exception****)*

un pezzo di codice cattura il segnale di eccezione per trattarla opportunamente

Trattamento *dell'eccezione (****handling exception****)*

- *informare l'utente dell'evento*
- *cerca eventualmente di riparare*
- *ripristinare uno stato consistente con la computazione in atto*

Eccezioni built-in

Java ha molte ***eccezioni predefinite****:*

AritmeticException *divisione per zero*

NullPointerException *tentativo di accedere a un null pointer*

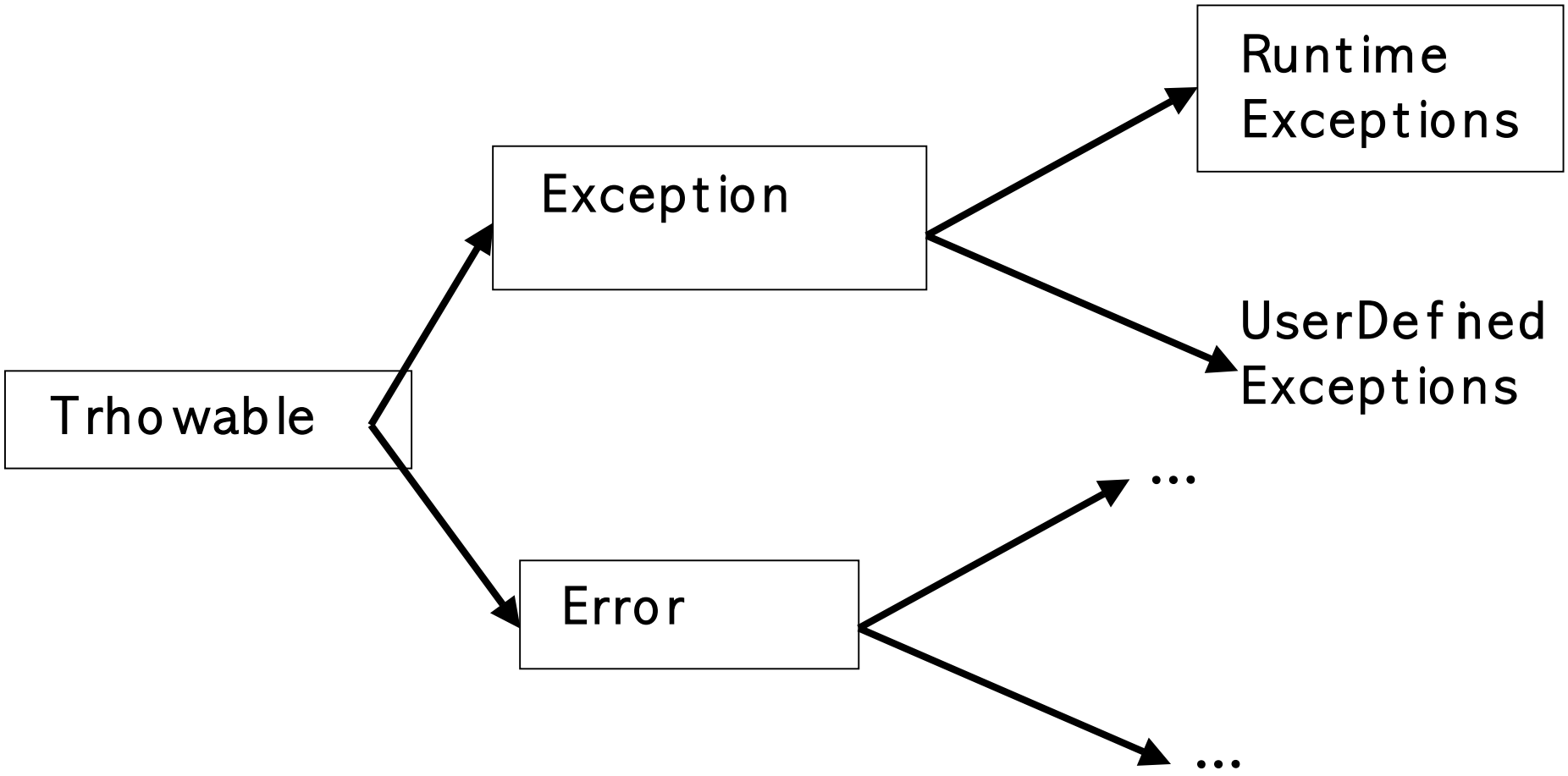
ClassCastException *tentativo di fare un cast che non rispetta vincoli imposti dal subtyping*

ArrayIndexOutOfBoundsException *accesso errato all'elemento di un array*

ClassNotFoundException *Il class loader richiede il caricamento di una classe che non viene trovata*

In Java le ***eccezioni sono oggetti!***

Gerarchia di Eccezioni



Eccezioni in Java: throw

In Java un eccezione viene sollevata attraverso la keyword **throw**.

L'eccezione deve essere prima **creata** (**è un oggetto**)

```
throw new NullPointerException()
```

Le classi di eccezioni predefinite **hanno due costruttori:**

- uno senza argomenti
- uno che riceve una stringa come argomento

Quindi possiamo creare un eccezione con un messaggio che informa sulla natura dell'eccezione o sul punto in cui è sollevata:

```
throw new Error("WrongWay")
```

Eccezioni ed esecuzione

Il sollevamento di un eccezione ***modifica il flusso di esecuzione****:*

L'esecuzione del metodo dove è stata lanciata l'eccezione viene abortita

Il supporto run-time cerca un ***exception handler***

La ricerca dell'exception handler può causare la terminazione dell'esecuzione di altri metodi sullo stack di attivazione

*Se la ricerca fallisce, l'eccezione sfugge (****escapes****) e causa la terminazione di tutto il programma (o il thread!)*

Trattamento delle Eccezioni

Il trattamento delle eccezioni avviene attraverso un blocco `try-catch`

Sintassi generale:

```
try{
// codice che potrebbe generare eccezioni
}
catch (E1 e)
  { /* handler per eccezione tipo E1 */ }
catch (E2 e)
  { // handler per eccezione tipo E2 */ }
...
catch (En e)
  { // handler per eccezione tipo En */ }
```

Trattamento delle Eccezioni

Si esegue il codice dentro il blocco `try{}` *.*

Se ***non sono sollevate eccezioni*** ***il blocco termina normalmente*****.**

Altrimenti, supponiamo venga lanciata l'eccezione `exc` *di tipo* ***E*** *all'interno del blocco* ***try****.*

Se $E <: E_1$ *allora si istanzia* ***e*** *con* ***exc*** *e si esegue il codice del primo handler, altrimenti…*

… *se* $E <: E_2$ *allora si istanzia* ***e*** *con* ***exc*** *e si esegue il codice del secondo handler, altrimenti…*

continua fino a quando sono stati esaminati tutti i blocchi ***catch****.*

Se nessun blocco ***catch*** *tratta un supertipo di* ***E****, l'eccezione* ***sfugge*** *e il metodo termina e l'eccezione viene* ***passata al chiamante***

Trattamento delle Eccezioni

I blocchi `try catch` *possono definire un numero arbitrario di handlers, purchè ciascuno associato a* ***tipi diversi*** *di eccezioni.*

Siccome vengono analizzati in ordine testuale è opportuno mettere ***prima quelli più specifici*** *(sottoclassi) e poi i più generali.*

Esempio (1)

I blocchi `try-catch` che non catturano eccezioni vengono ignorati:

```
try{
  throw new NumberFormatException();
}
catch (ArrayIndexOutOfBounds e)
  { System.out.println("Array exception");}
catch (NumberFormatException e)
  {System.out.println("Number exception");}
}
```

Output: `Number exception`

Esempio (2)

Se nessun handler type fa il match, l'eccezione sfugge:

```
try{
  throw new NumberFormatException();
}
catch (ArrayIndexOutOfBounds e)
  { System.out.println("Array exception");}
catch (ClassCastException e)
  {System.out.println("Class exception");}
}
```

Output: `Exception in thread "main"`
`Java.lang.NumberFormatException`

Esempio (3)

Il match, può avvenire a causa di subtyping:

```
try{
  throw new NumberFormatException();
}
catch (ArrayIndexOutOfBounds e)
  { System.out.println("Array exception");}
catch (Exception e)
  {System.out.println("General exception");}
}
```

Output: `General Exception`

Esempio (4)

La clausola `catch` si applica solo alle eccezioni generate dal codice protetto dal blocco `try`:

```
try{
  throw new ArrayIndexOutOfBounds();
}
catch (ArrayIndexOutOfBounds e)
  {throw new ArrayIndexOutOfBounds();}
catch (Exception e)
  {System.out.println("Number exception");}
}
```

Output: `Exception in thread "main"`
`java.lang.ArrayOutOfBoundsException`

Finally

Permette di definire un blocco che viene sempre e comunque eseguito, indipendentemente dall'eccezione lanciata:

```
try{
  parcheggiaMotorino();
}
finally {mettiLaCatena();}
```

Esempio: ***Un file aperto deve comunque essere chiuso indipendentemente dall'esito della computazione***

Eccezioni user-defined

Le eccezioni possono essere dichiarate.

Le eccezioni sono classi che estendono `Trowable`.

```
class ServerTimeOutException extends
    Exception{
private int port;
public ServerTimeOutException(String
    reason, int port) {super(reason);
    this.port=port;}
public int port() {return port;}
}
```

Eccezioni checked/unchecked

Le eccezioni checked sono eccezioni che a cui di solito si può riparare.

`Throwable, Exception` *e sottoclassi vanno gestite e dichiarate*

Le eccezioni unchecked sono eccezioni che di solito sono irreparabili.

`Error, RunTimeException` *e sottoclassi non devono essere gestite e dichiarate*

Eccezioni checked

```
class MyException extends Exception{...}
```

Non compila:

```
public void m(){throw new MyException();}
```

Compila:

```
public void m() throws MyException
    {throw new MyException();}
```

Compila:

```
public void m() {
    try{ throw new MyException();}
    catch (MyExceptione)
    {System.out.println("tratto");}}
```

Eccezioni unchecked

```
class MyException extends Error{...}
```

Compila:

```
public void m(){throw new MyException();}
```

Compila:

```
public void m() throws MyException
    {throw new MyException();}
```

Compila:

```
public void m() {
    try{ throw new MyException();}
    catch (MyException
    e){System.out.println("tratto")}}
```

Quando usare eccezioni

*Se il contesto di uso di un metodo è locale (**privato**) forse è meglio **non usare eccezioni** (costose)*

*Le eccezioni **unchecked** vanno usate quando ci si aspetta che **non siano usualmente sollevate**.*

Sarebbe opportuno sollevare un'eccezione se i parametri non soddisfao **`PREC`**

Come comportarsi...(1)

Reflecting: *un'eccezione viene semplicemente* ***rilanciata*** *al chiamante del metodo dove è stata sollevata l'eccezione.*

Può avvenire in due modi:

automatico*: è quello che succede normalmente alle eccezioni non trattate*

esplicito*: il programmatore può esplicitamente riflettere un'eccezione, tipicamente* ***creandone una di tipo diverso***

Come comportarsi…(2)

Masking: *un'eccezione catturata all'interno di un metodo viene intercettata e trattata opportunamente*

Le eccezioni unchecked ***andrebbero trattate e mascherate*** *(i metodi non sono obbligati a definire che le possono sollevare), nella porzione di codice più piccola che le può sollevare.*

Osservazione: *Un errore a un certo livello di astrazione potrebbe non esserlo a un altro livello*

Esempio: reflecting

```
public class Arrays{
  public static int min(int [] a, int x)
    throws NullPointerException,
    EmptyException {
/* POST:se a==NULL solleva NullPointerException
 * se a è vuoto solleva EmptyException
 * altrimenti torna il valore minimo di a */
int m;
try {m=a[0];}
    catch(IndexOutOfBoundsException)
      {throw new EmptyException("Arrays.min");}
    …}
```

Osservate che la nuova eccezione cerca di esprimere il significato logico del comportamento anomalo!

Esempio: masking

```
public class Arrays{
  public static boolean sorted(int [] a)
    throws NullPointerException
/* POST:se a==NULL solleva NullPointerException
 * se a è ordinato in modo crescente torna TRUE
 * altrimenti FALSE */
int prev;
try {prev=a[0];}
    catch(IndexOutOfBoundsException)
      {return true;}
  … }
```

In questo caso l'eccezione viene ricondotta a una "normale" esecuzione del metodo

Esempio: verifica PREC

```
public static int search(int [] a, int x)
    throws NullPointerException,
      NotFoundException
/* PREC: a è un vettore ordinato
 * POST: se a è NULL solleva
   NullPointerException
 * se x non occorre in a, solleva
   NotFoundException
 * altrimenti torna i, a[i]==x
 */
```

Verificare PREC può essere oltre che time-consuming durante la programmazione, oneroso in termini di efficienza

Defensive Programming

*Scrivere procedure che si **difendono contro gli errori** introdotti da altre procedure, hardware, inserimento dati utente, network etc.*

*Cercare il più possibile di fare **recover** a uno **stato coerente di computazione**.*

*Promuovere una eventuale **graceful degradation***