Programmazione a Oggetti
Lezione 11

Eccezioni e Packages

Sommario

- Eccezioni
  - Sollevamento
  - Cattura
  - Trattamento
- Packages e visibilità

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

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

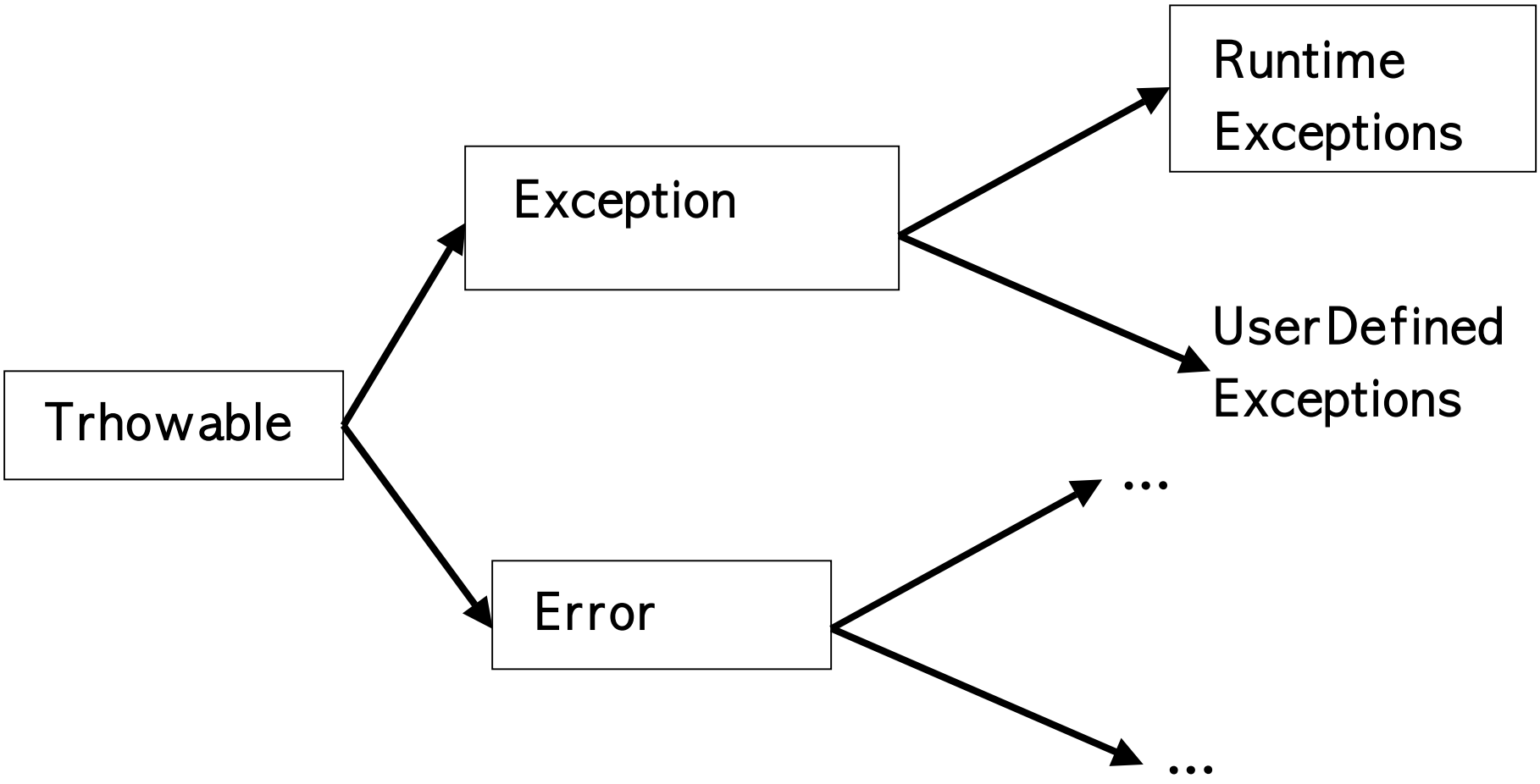
ClassCastException tentativo di fare un cast che non rispetta vincoli imposti dal subtyping

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:

```
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 $\texttt{E} <: \texttt{E}_1$ allora si istanzia `e` con `exc` e si esegue il codice del primo handler, altrimenti...

... se $\texttt{E} <: \texttt{E}_2$ 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 (MyException
    e){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")}}
```

Packages

Costrutto per assemblare insiemi di classi e interfacce.

Si crea una directory `myPackage`

Si copiano tutti i file *.java in questa directory

Prima linea di ogni file: `package myPackage;`

Si dichiarano pubbliche tutte le classi che si vogliono rendere visibili ai clienti del package.

Nota: solo una classe per file può essere pubblica

Uso dei packages

Un package si importa usando la direttiva:

```
import myPackage.*
```

Vengono rese visibili tutte le classi e interfacce dichiarate `public` in `myPackage`.

Ricordare il livello di accesso package (è il default!)

Protected: un'anomalia (!?)

Package Greek:

```
public class Alpha{
 protected void protM()
    {System.out.println("protMtd");}
}
class Beta extends Alpha {
 void accessM(Alpha a){a.protM(); protM();}
}
```

Package Latin:

```
Import Greek.*
class Delta extends Alpha {
 void accessM(Alpha a, Delta d)
    {a.protM(); d.protM();}
}
```

Type error

Protected: un'anomalia (!?)

Meccanismo corretto: ci sono due gerarchie che hanno come avo comune la classe `Alpha`.

`Delta` e `Beta` sono estranee, e la visibilità protected garantische che non possono accedere ai membri ereditati da `Alpha`.

Il parametro `Alpha a` potrebbe avere tipo dinamico `Beta` e il metodo protected `protM()` potrebbe essere ridefinito in `Beta`.