Linguaggi di Programmazione I – Lezione 15

Prof. Marcello Sette
mailto://marcello.sette@gmail.com
http://sette.dnsalias.org

5 giugno 2008

Eccezioni (Gestione degli errori)

Eccezioni: il meccanismo 3 / 38

Introduzione

- Le `eccezioni` denotano "eventi eccezionali" la cui occorrenza altera il flusso normale delle istruzioni.
- Es.: risorse hardware indisponibili, hardware malfunzionante, bachi nel software ...
- Quando capita un tale evento, si dice che viene "lanciata una eccezione".

`try` e `catch`

- Il codice che potrebbe lanciare una eccezione è inglobato in un blocco marcato `try`.
- Il codice che assume la responsabilità di fare qualcosa in conseguenza del lancio di una eccezione si chiama *manipolatore* (exception handler) e va inglobato in una clausola `catch`.

```
try {
  // Qui va scritto il codice "rischioso"
}
catch(Eccezione1 e) {
  // Qui il codice che manipola una Eccezione1
}
catch(Eccezione2 e) {
  // Qui il codice che manipola una Eccezione2
}
// codice non rischioso va scritto qui
```

`finally`

- Un blocco opzionale marcato `finally` verrà (quasi) SEMPRE eseguito, anche dopo il lancio ed la manipolazione (eventuale) dell'eccezione.

```
try {
  // Qui va scritto il codice "rischioso"
}
catch(Eccezione1 e) {
  // Qui il codice che manipola una Eccezione1
}
catch(Eccezione2 e) {
  // Qui il codice che manipola una Eccezione2
}
finally {
  // Codice da eseguire in ogni caso
}
// codice non rischioso va scritto qui
```

- Il blocco `finally` viene eseguito perfino dopo una eventuale istruzione `return` presente nei blocchi `try` o `catch`.
- IL BLOCCO `finally` VIENE ESEGUITO SEMPRE.
- Il blocco `finally` potrebbe non essere eseguito o potrebbe non completare l'esecuzione solo in conseguenza di un crash totale del sistema oppure tramite una invocazione di `System.exit(int status)`.

Vincoli

- Le clausole `catch` ed il blocco `finally` sono opzionali.
- Dopo un blocco `try` deve esistere almeno una clausola `catch` oppure un blocco `finally`. Un blocco `try` solitario causa un errore di compilazione.
- Se esistono una o più clausole `catch` esse devono seguire immediatamente il blocco `try`.
- Se esiste il blocco `finally` esso deve seguire l'ultima clausola `catch`.
- Non è ammessa nessuna istruzione tra il blocco `try`, le clausole `catch` ed il blocco `finally`.
- È significativo l'ordine in cui si succedono tra loro le clausole `catch` (vedremo tra poco).

Propagazione

- Perché le clausole `catch` non sono obbligatorie?
- Che succede ad una eccezione che viene lanciata da un blocco `try`, ma per la quale non esiste una clausola `catch` che l'attende?
- Una eccezione non catturata semplicemente "si immerge" nel record di attivazione che la ha generata, "riemergendo" nel successivo record di attivazione.
- Qui potrebbe essere catturata da una ulteriore clausola `catch`, oppure altrimenti ricadere nel record successivo, e così via. Finché, eventualmente, l'eccezione raggiunge il record di attivazione del `main`, dal quale, se non catturata, "esplode", producendo (forse) una pittoresca descrizione del proprio percorso (stack trace).

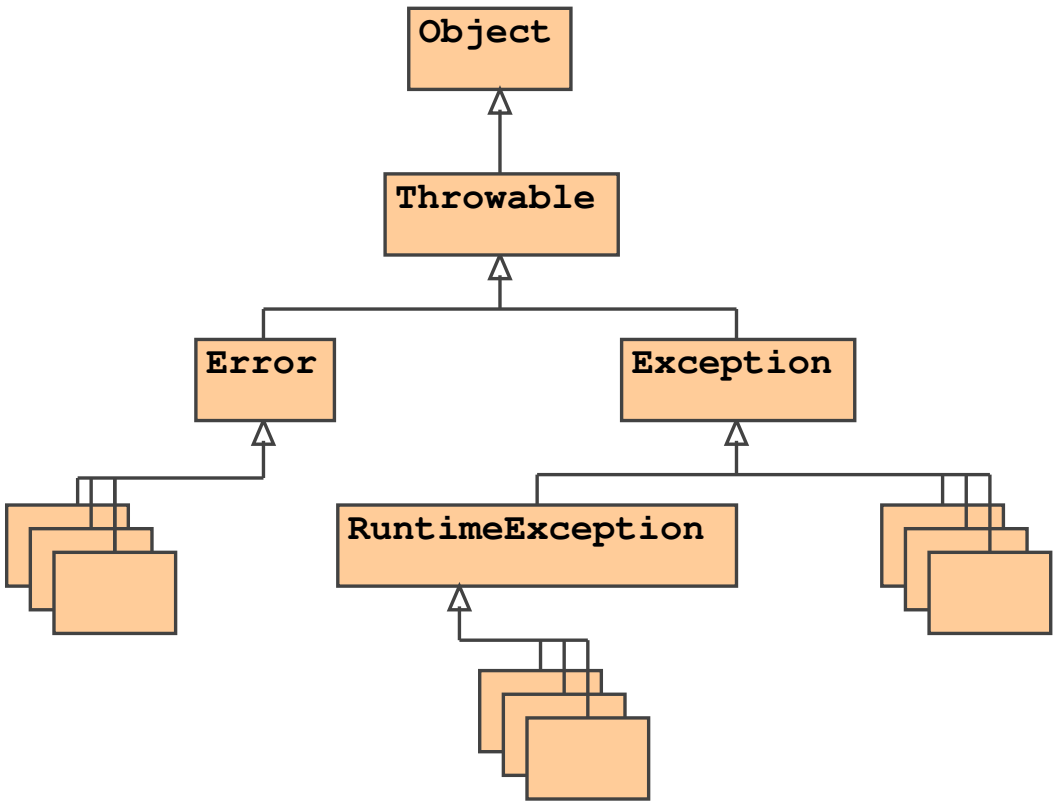
Definizione

- In Java tutto ciò che non è primitivo è un oggetto. Le eccezioni non fanno "eccezione" a questa regola.
- Ogni eccezione è una istanza di una sotto-classe della classe `Exception`.
- Una eccezione viene lanciata usando la parola riservata `throw`:

```
throw new Exception();
```

- Tutto ciò che segue, nello stesso blocco di istruzioni, il lancio dell'eccezione non verrà eseguito (tranne l'eventuale blocco `finally`).
- La gerarchia degli oggetti lanciabili è la seguente:

Gerarchia (1)



Gerarchia (2)

- La classe `Throwable` rappresenta tutti gli oggetti che possono essere lanciati. Essa contiene il metodo `printStackTrace`.
- La classe `Error` e le sue sottoclassi rappresentano situazioni insolite che non sono causate da errori di programmazione o da ciò che normalmente succede durante l'esecuzione del programma. Per esempio, la JVM ha esaurito la memoria oppure qualche altra risorsa non è disponibile. In genere, una applicazione non deve essere capace di riprendersi da una situazione di errore. Pertanto, un programma non è obbligato a gestire gli oggetti `Error`: esso compila senza problemi.
- La classe `RuntimeException` rappresenta pure eventi eccezionali, ma dovuti al programma (errori di programmazione, bachi). Sono qui evidenziati perché indicano eccezioni rare e difficili da gestire (discuteremo in seguito). Il programmatore, che si accorge di un baco dovuto ad un suo errore, deve correggerlo, non gestirlo come una eccezione!

Eccezioni catturate (1)

- Una clausola `catch` cattura ogni oggetto-eccezione il cui tipo può essere ricondotto mediante conversione automatica al tipo specificato nella clausola.
- Esempio: la classe `IndexOutOfBoundsException` ha due sottoclassi, `ArrayIndexOutOfBoundsException` e `StringIndexOutOfBoundsException`; si può scrivere una unica clausola che catturi una qualunque di queste eccezioni:

```
try {
  // Codice che potrebbe lanciare una eccezione
  // IndexOutOfBoundsException oppure
  // ArrayIndexOutOfBoundsException oppure
  // StringIndexOutOfBoundsException
}
catch (IndexOutOfBoundsException e) {
  e.printStackTrace();
}
```

Eccezioni catturate (2)

- Resistere alla tentazione di scrivere una unica clausola catch-all:

```
try {
  // codice rischioso
}
catch (Exception e) {
}
```

- L'ordine delle clausole `catch` è importante.
- Nell'esempio precedente, se avessimo scritto:

```
try {
  // Codice che potrebbe lanciare una eccezione
  // IndexOutOfBoundsException oppure
  // ArrayIndexOutOfBoundsException
}
catch (IndexOutOfBoundsException e) {
  // tratta l'eccezione
}
catch (ArrayIndexOutOfBoundsException e) {
  // tratta l'eccezione
}
```

il codice non sarebbe stato compilato.

Eccezioni catturate (3)

- È corretto, invece, scrivere:

```
try {
  // Codice che potrebbe lanciare una eccezione
  // IndexOutOfBoundsException oppure
  // ArrayIndexOutOfBoundsException
}
catch (ArrayIndexOutOfBoundsException e) {
  // tratta l'eccezione
}
catch (IndexOutOfBoundsException e) {
  // tratta l'eccezione
}
```

. . . e non catturate

- Come facciamo a sapere che un metodo può lanciare una eccezione che dobbiamo catturare?
- Così come la dichiarazione del metodo deve specificare il numero e il tipo dei parametri, il tipo di ritorno, anche le eccezioni che un metodo può lanciare DEVONO essere dichiarate (a meno che non siano sottoclassi di `RuntimeException`).
- La parola chiave `throws` viene usata per elencare le eccezioni che possono fuoriuscire da un metodo:

```
void miaFunzione() throws MiaEccezione1, MiaEccezione2 {
  // qui il codice per il metodo
}
```

- Il fatto che un metodo dichiara l'eccezione non significa che esso la lancerà sempre, ma avverte l'utilizzatore che esso potrebbe lanciarla.

Handle or Declare

- Se un metodo non lancia direttamente una eccezione, ma richiama un altro metodo che può farlo, allora si deve scegliere almeno una di queste opzioni (regola *handle or declare*):
  1. Gestire l'eccezione fornendo le opportune sezioni `try/catch`.
  2. Propagare l'eccezione dichiarandola nell'intestazione del metodo.
- Una "eccezione" a questa regola: le `RuntimeException` sono esenti dall'obbligo di dichiarazione. Esse sono *unchecked* dal compilatore, mentre le rimanenti eccezioni sono dette *checked*.
  Ora, forse, si capisce il motivo della gerarchia precedentemente esposta.
- Nota: l'*or* della regola non è esclusivo, nel senso che si può decidere di fare entrambe le cose.

Esempio (1)

Quali problemi ci sono in questo codice?

```
void f1() {
  f2();
}

void f2() {
  throw new IOException();
}
```

- Il metodo `f2` lancia una eccezione `checked` ma non la dichiara;
- Se esso l'avesse dichiarata, come in:

```
void f2() throws IOException {...}
```

  il problema l'avrebbe `f1` che dovrebbe ora dichiararla o catturarla.

Esempio (2)

Quali problemi ci sono in questo codice?

```
import java.io.*;
class Test {
  public int f1() throws EOFException {
    return f2();
  }
  public int f2() throws EOFException {
    // qui il codice che lancia effettivamente l'eccezione
    return 1;
  }
}
```

- Poiché `EOFException` è sottoclasse di `IOException`, che è sottoclasse di `Exception`, essa è una eccezione *checked*. Essa viene regolarmente dichiarata ed il codice regolarmente compilato.

Esempio (3)

Quali problemi ci sono in questo codice?

```
public void  f1() {
  // qui codice che puo' lanciare NullPointerException
}
```

- Poiché `NullPointerException` è sottoclasse di `RuntimeException`, essa è una eccezione *unchecked*. Non è necessario nè dichiararla, nè catturarla, ed il codice viene regolarmente compilato.

Esempio (4)

Analogamente, questo codice compila correttamente:

```
class TestEx {
  public static void main (String [] args) {
    mioMetodo();
  }
  static void mioMetodo() { // Non c'e' bisogno di
                            // dichiarare un Error
    fai();
  }
  static void fai() { // Non c'e' bisogno di dichiarare un Error
    try {
      throw new Error();
    }
    catch(Error me) {
      throw me; // Ti ho preso, ma ora di rilancio
    }
  }
}
```

Nuove eccezioni

- È possibile usare tipi di eccezioni già presenti nelle Java API, oppure crearne di propri in questo modo:

```
class MiaEccezione extends Exception { }
```

  oppure estendendo una qualunque sottoclasse di `Exception`.
- Da questo momento in poi si può lanciare un oggetto del tipo (checked) `MiaEccezione`.
- Pertanto, il codice seguente non compila:

```
class TestEx {
  void f() {
    throw new MiaEccezione();
  }
}
```


Overriding (ancora?)

Posto che sono sovrapponibili solo i metodi visibili della superclasse, ecco finalmente le **regole complete per la sovrapposizione di metodi**:

- I due metodi devono avere identica segnatura.
- I due metodi devono avere identico tipo di ritorno.
- Non si può marcare `static` uno solo dei metodi.
- Il metodo nella superclasse non può essere marcato `final`.
- Il metodo nella sottoclasse deve avere visibilità non inferiore a quello della superclasse.
- Il metodo nella sottoclasse può dichiarare di lanciare un tipo di eccezione *checked*, ma tale tipo non deve essere un nuovo tipo o un tipo più "esteso" rispetto a quelli dichiarati dal metodo della superclasse.
  Cioè, le EVENTUALI eccezioni *checked* dichiarate dal metodo nella sottoclasse, DEVONO essere tipi posti al di sotto nella gerarchia delle eccezioni dichiarate dal metodo nella superclasse.

Esercizi 24 / 38

Esercizi

Sono proposti in allegato due esercizi:

1. In questo esercizio si dovranno usare i blocchi `try-catch` per gestire una semplice `RuntimeException`.
2. In questo esercizio si dovrà creare il nuovo tipo `OverdraftException` di oggetti lanciabili dal metodo `preleva` nella classe `Conto`.

D 1

Dato il codice seguente:

```
 1. try {
 2.   // codice rischioso; hp:
 3.   // Exception
 4.   //   +--- EccA
 5.   //          +--- EccB
 6    //          +--- EccC
 7.   System.out.print(1);
 8. }
 9. catch (EccB e) {
10.   System.out.println(2);
11. }
12. catch (EccC e) {
13.   System.out.println(3);
14. }
15. catch (Exception e) {
16.   System.out.println(4);
17. }
18. finally {
19.   System.out.println(5);
20. }
21. System.out.println(6);
```

Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 venga lanciata una eccezione di tipo EccB?

A. 1
B. 2
C. 3
D. 4
E. 5
F. 6

D 2

Dato il codice seguente:

```
 1. try {
 2.   // codice rischioso; hp:
 3.   // Exception
 4.   //   +--- EccA
 5.   //          +--- EccB
 6    //          +--- EccC
 7.   System.out.print(1);
 8. }
 9. catch (EccB e) {
10.   System.out.println(2);
11. }
12. catch (EccC e) {
13.   System.out.println(3);
14. }
15. catch (Exception e) {
16.   System.out.println(4);
17. }
18. finally {
19.   System.out.println(5);
20. }
21. System.out.println(6);
```

Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 non venga lanciata alcuna eccezione?

A. 1
B. 2
C. 3
D. 4
E. 5
F. 6

D 3

Dato il codice seguente:

```
 1. try {
 2.   // codice rischioso; hp:
 3.   // Exception
 4.   //   +--- EccA
 5.   //          +--- EccB
 6    //          +--- EccC
 7.   System.out.print(1);
 8. }
 9. catch (EccB e) {
10.   System.out.println(2);
11. }
12. catch (EccC e) {
13.   System.out.println(3);
14. }
15. catch (Exception e) {
16.   System.out.println(4);
17. }
18. finally {
19.   System.out.println(5);
20. }
21. System.out.println(6);
```

Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 venga lanciata una eccezione di tipo `EccC`?

A. 1
B. 2
C. 3
D. 4
E. 5
F. 6

D 4

Dato il codice seguente:

```
 1. try {
 2.   // codice rischioso; hp:
 3.   // Exception
 4.   //   +--- EccA
 5.   //          +--- EccB
 6    //          +--- EccC
 7.   System.out.print(1);
 8. }
 9. catch (EccB e) {
10.   System.out.println(2);
11. }
12. catch (EccC e) {
13.   System.out.println(3);
14. }
15. catch (Exception e) {
16.   System.out.println(4);
17. }
18. finally {
19.   System.out.println(5);
20. }
21. System.out.println(6);
```

Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 venga lanciata una eccezione di tipo `EccA`?

A. 1
B. 2
C. 3
D. 4
E. 5
F. 6

D 5

Dato il codice seguente:

```
 1. try {
 2.   // codice rischioso; hp:
 3.   // Exception
 4.   //   +--- EccA
 5.   //          +--- EccB
 6    //          +--- EccC
 7.   System.out.print(1);
 8. }
 9. catch (EccB e) {
10.   System.out.println(2);
11. }
12. catch (EccC e) {
13.   System.out.println(3);
14. }
15. catch (Exception e) {
16.   System.out.println(4);
17. }
18. finally {
19.   System.out.println(5);
20. }
21. System.out.println(6);
```

Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 venga lanciata una eccezione di tipo `Exception`?

A. 1
B. 2
C. 3
D. 4
E. 5
F. 6

D 6

Dato il codice seguente:

```
 1. try {
 2.   // codice rischioso; hp:
 3.   // Exception
 4.   //   +--- EccA
 5.   //          +--- EccB
 6    //          +--- EccC
 7.   System.out.print(1);
 8. }
 9. catch (EccB e) {
10.   System.out.println(2);
11. }
12. catch (EccC e) {
13.   System.out.println(3);
14. }
15. catch (Exception e) {
16.   System.out.println(4);
17. }
18. finally {
19.   System.out.println(5);
20. }
21. System.out.println(6);
```

Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 venga lanciata una eccezione di tipo `RuntimeException`?

A. 1
B. 2
C. 3
D. 4
E. 5
F. 6

D 7

Dato il codice seguente:

```
 1. try {
 2.   // codice rischioso; hp:
 3.   // Exception
 4.   //   +--- EccA
 5.   //          +--- EccB
 6    //          +--- EccC
 7.   System.out.print(1);
 8. }
 9. catch (EccB e) {
10.   System.out.println(2);
11. }
12. catch (EccC e) {
13.   System.out.println(3);
14. }
15. catch (Exception e) {
16.   System.out.println(4);
17. }
18. finally {
19.   System.out.println(5);
20. }
21. System.out.println(6);
```

Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 venga lanciata una eccezione di tipo `Error`?

A. 1
B. 2
C. 3
D. 4
E. 5
F. 6

D 8

```
public class M {
  public static void
      main(String[] args) {
    int k=0;
    try {
      int i=5/k;
    }
    catch (ArithmeticException e) {
      System.out.print(1);
    }
    catch (RuntimeException e) {
      System.out.print(2);
      return;
    }
    catch (Exception e) {
      System.out.print(3);
    }
    finally {
      System.out.print(4);
    }
    System.out.print(5);
  }
}
```

Qual è l'output del programma precedente?

A. 5
B. 14
C. 124
D. 145
E. 1245
F. 35

D 9

```
public class Eccezioni {
  public static void main(String[] args) {
    try {
      if (args.length == 0) return;
      System.out.println(args[0]);
    }
    finally {
      System.out.println("Fine");
    }
  }
}
```

Quali affermazioni riguardanti il precedente programma sono vere?

A. Se eseguito senza argomenti, il programma non produce output.
B. Se eseguito senza argomenti, il programma stampa `Fine`.
C. Il programma lancia un `ArrayIndexOutOfBoundsException`.
D. Se eseguito con un argomento, il programma stampa solo l'argomento dato.
E. Se eseguito con un argomento, il programma stampa l'argomento dato seguito da `Fine`.

D 10

Qual è l'output del seguente programma?

```
public class MyClass {
  public static void main(String[] args) {
    RuntimeException re = null;
    throw re;
  }
}
```

A. Il codice non viene compilato, poiché il `main` non dichiara che lancia una `RuntimeException`.
B. Il codice non viene compilato, poiché non può rilanciare `re`.
C. Il programma viene compilato e lancia `java.lang.RuntimeException` in esecuzione.
D. Il programma viene compilato e lancia `java.lang.NullPointerException` in esecuzione.
E. Il programma viene compilato, eseguito e termina senza produrre alcun output.

D 11

Quali di queste affermazioni sono vere?

A. Se una eccezione non è catturata in un metodo, il metodo termina e viene ripresa la successiva normale esecuzione.
B. Un metodo sovrapposto in una sottoclasse deve dichiarare che lancia lo stesso tipo di eccezione del metodo che sovrappone.
C. Il `main` può dichiarare che lancia eccezioni `checked`.
D. Un metodo che dichiara di lanciare una certo tipo di eccezione, può lanciare una istanza di una qualunque sottoclasse di quel tipo.
E. Il blocco `finally` è eseguito se e solo se viene lanciata una eccezione all'interno del corrispondente blocco `try`.

D 12

```
public class MiaClasse {
  public static void main
           (String[] args} {
    try {
      f();
    }
    catch (MiaEcc e) {
      System.out.print(1);
      throw new RuntimeException();
    }
    catch (RuntimeException e) {
      System.out.print(2);
      return;
    }
    catch (Exception e) {
      System.out.print(3);
    }
    finally {
      System.out.print(4);
    }
    System.out.print(5);
  }

  // MiaEcc e'
  // sottoclasse di Exception
  static void f() throws MiaEcc {
    throw new MiaEcc();
  }
}
```

Qual è l'output del programma precedente?

A. 5
B. 14
C. 124
D. 145
E. 1245
F. 35

D 13

```
public class MiaClasse {
  public static void main
              (String[] args)
                throws MiaEcc {
    try {
      f();
      System.out.print(1);
    }
    finally {
      System.out.print(2);
    }
    System.out.print(3);
  }

  // MiaEcc e'
  // sottoclasse di Exception
  static void f() throws MiaEcc {
    throw new MiaEcc();
  }
}
```

Qual è l'output del programma precedente?

A. 2 e lancia `MiaEcc`.
B. 12
C. 123
D. 23
E. 32
F. 13