Programmazione II

A.A. 2022-23
Prof. Maria Tortorella

Exception Handling

- Lanciare un'eccezione
- Progettare una classe eccezione
- Catturare un'eccezione

Condizioni di Errore

- Una condizione di errore in un programma può avere molte cause

- Errori di programmazione
  - Divisione per zero, cast non permesso, accesso oltre i limiti di un array, …
- Errori di sistema
  - Disco rotto, connessione remota chiusa, memoria non disponibile, …
- Errori di utilizzo
  - Input non corretti, tentativo di lavorare su file inesistente, …

Condizioni di Errore in java

- Java ha una gerarchia di classi per rappresentare le varie tipologie di errore
  - dislocate in package diversi a seconda del tipo di errore
- La superclasse di tutti gli errori è la classe **`Throwable`** nel package **`java.lang`**.

- Qualsiasi nuovo tipo di errore deve essere inserito nella discendenza di **`Throwable`**
  - Le parole chiave di java per la gestione degli errori possono essere usate solo su oggetti di questa classe.

Gestione delle eccezioni

- Tradizionalmente:
  - Il metodo (funzione) restituisce un codice che rappresenta l'errore.
    - un metodo che cerca un oggetto in un array, restituisce -1 se non trova l'oggetto
- Problema:
  - Il programma chiamante non verifica il codice restituito da un metodo
    - La notifica del problema viene persa
- Problema:
  - Il chiamante può non essere in grado di gestire il problema
    - Il programma dovrebbe fallire . . . .

Gestione delle eccezioni

- Nella gestione tradizionale, anziché focalizzarsi sul da farsi, sulle operazioni che devono essere eseguite

```
x.doSomething()
```

  - Ci si focalizza sui possibili fallimenti che possono avvenire:

```
if (!x.doSomething()) return false;
```

  - La gestione delle eccezioni permette di focalizzarsi sul da farsi, demandando ad un programma apposito, il gestore delle eccezioni, il controllo dei fallimenti

Lanciare una eccezione

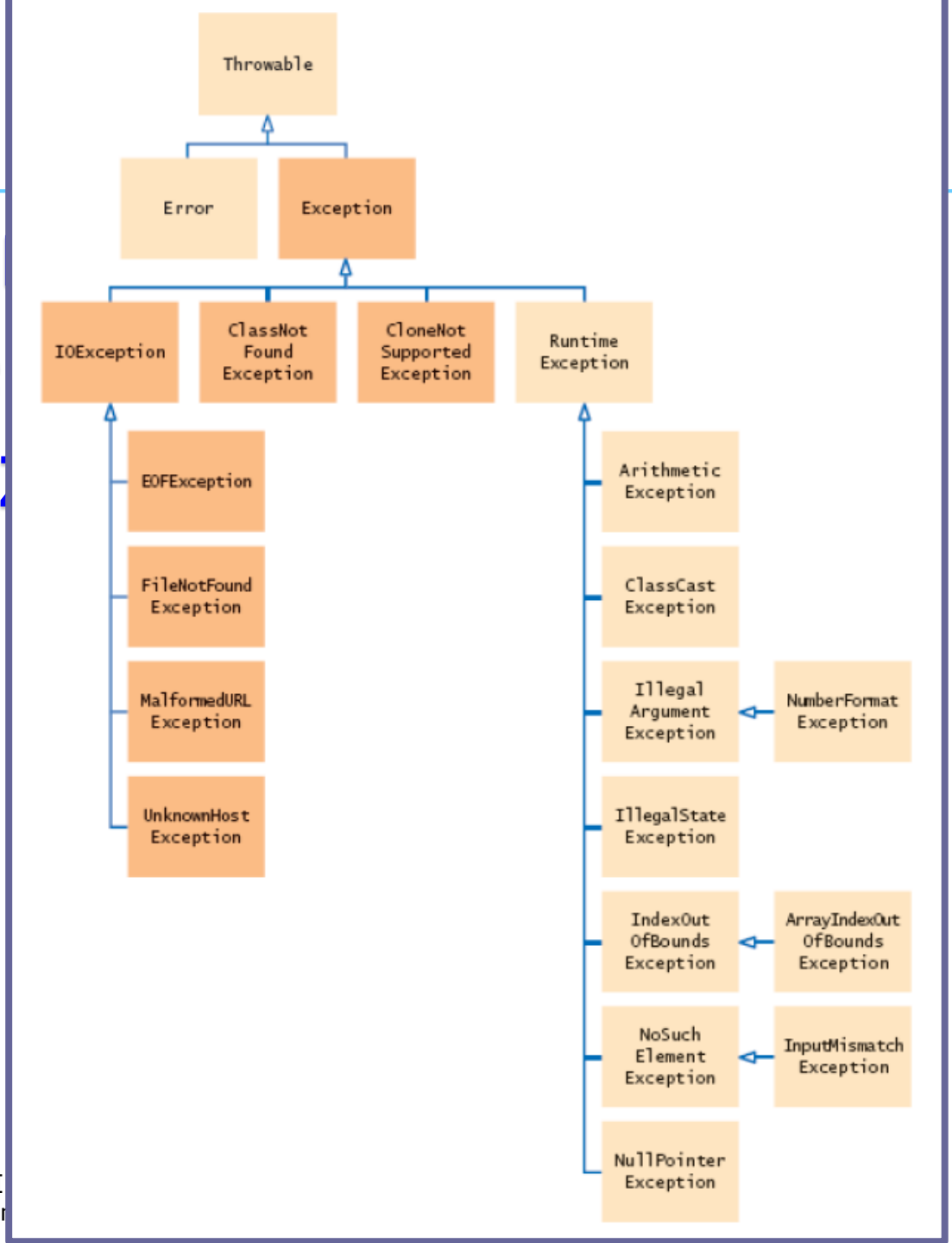
- Eccezioni:
  - Non necessariamente errori . . .
- Una eccezione è un evento che interrompe la normale esecuzione del programma
- Se si verifica un'eccezione il metodo trasferisce il controllo ad un gestore delle eccezioni
  - La gestione delle eccezioni è un meccanismo per trasferire il controllo dell'esecuzione del programma dal punto in cui viene segnalato l'errore ad un apposito handler (un gestore per il ripristino della situazione dell'errore)

Eccezioni

- Java mette a disposizione varie classi per gestire le eccezioni. Tra i package che contengono queste classi, ci sono:
  - java.lang
  - java.io

- Tutte le classi che gestiscono le eccezioni sono ereditate dalla classe `Exception`

Gera
de
ecce


Throwable
Error
Exception
IOException
ClassNot Found Exception
CloneNot Supported Exception
Runtime Exception
EOFException
FileNotFound Exception
MalformedURL Exception
UnknownHost Exception
Arithmetic Exception
ClassCast Exception
Illegal Argument Exception
NumberFormat Exception
IllegalState Exception
IndexOut OfBounds Exception
ArrayIndexOut OfBounds Exception
NoSuch Element Exception
InputMismatch Exception
NullPointer Exception

Eccezioni Checked e Unchecked

- Checked
  - Il compilatore si fa carico di verificare che non siano ignorate
  - Sono dovute a circostanze esterne che il programmatore non può prevenire
  - Esempi tipici si verificano con le operazioni di input e output
    - **`IOException`**: terminazione dovuta ad un'errata operazione di input-output
    - **`EOFException`**: terminazione inaspettata del flusso di dati in ingresso
    - Possono essere provocate da eventi esterni
      - errore del disco
      - interruzione del collegamento di rete
    - Il gestore dell'eccezione si occupa del problema
- Il compilatore richiede che sia aggiunto `throws Exception` nella intestazione dei metodi

Eccezioni Checked e Unchecked

- Unchecked:
  - Estendono la classe **`RuntimeException`** or **`Error`**
  - Derivano da problemi di programmazione che si possono evitare, correggendo il programma
  - Esempi di runtime exception:
  - **`NullPointerException`**: uso di un riferimento null
  - **`IndexOutofBoundException`**: accesso ad elementi esterni ai limiti di un array

- Non bisogna necessariamente installare un gestore per questo tipo di eccezione

Eccezioni controllate

- Tutte le sottoclassi di **`IOException`**
  - **`EOFException`**
  - **`FileNotFoundException`**
  - **`MalformedURLException`**
  - **`UnknownHostException`**
- **`ClassNotFoundException`**
- **`CloneNotSupportedException`**

Eccezioni non controllate

- Tutte le sottoclassi di RunTimeException
  - **`ArithmeticException`**
  - **`ClassCastException`**
  - **`IllegalArgumentException`**
  - **`IllegalStateException`**
  - **`IndexOutOfBoundsException`**
  - **`NoSuchElementException`**
  - **`NullPointerException`**

Eccezioni Checked e Unchecked

- Ci sono casi difficili da classificare:
  - `Scanner.nextInt` può lanciare una eccezione unchecked `InputMismatchException`
  - Il programmatore non può evitare che l'utente inserisca un dato non corretto
- E' importante gestire le eccezioni checked principalmente quando si lavora con file e streams

```
Scanner sc = new Scanner(…);
…
if (sc.hasNextInt())
      n = sc.nextInt();
else throw new Input…()
```

Lanciare una eccezione

- Java consente di lanciare un oggetto di tipo Exception per segnalare una condizione di esecuzione eccezionale
- Per lanciare un'eccezione, usiamo la parola chiave throw (lancia), seguita da un oggetto di tipo Exception
- Esempio: **`IllegalArgumentException`**

```
// illegal parameter value
IllegalArgumentException exception
   = new IllegalArgumentException("Amount exceeds balance");
throw exception;
```

Lanciare una eccezione

. . . . o alternativamente:

```
throw new IllegalArgumentException("Amount exceeds balance");
```

- Quando viene lanciata una eccezione il metodo termina immediatamente
  - Il controllo passa ad un exception handler
  - Le istruzioni successive non vengono eseguite

Esempio

```
public class BankAccount {
   public void withdraw(double amount){
      if (amount > balance){
           IllegalArgumentException exception
               = new IllegalArgumentException("Amount
                  exceeds balance");
           throw exception;
      }
      balance = balance - amount;
   }
   . . .
}
```

La stringa in input al costruttore di IllegalArgumentException rappresenta il messaggio d'errore da associare all'eccezione

Lanciare un'eccezione

- Consideriamo la catena di chiamate

  main → method1 → method2

  - Supponiamo che method2 incontra un imprevisto e lancia l'eccezione
    throw *riferimento ad oggetto Exception*
  - Il metodo in esecuzione termina immediatamente, l'Exception passa attraverso ogni invocazione della catena, forzando ogni metodo a terminare. Viene visualizzato:

    Some Exception
    at TryThrow.method2(TryThrow.java:18)
    at TryThrow.method1(TryThrow.java:15)
    at TryThrow.main(TryThrow.java:12)

Nota sintattica

```
throw exceptionObject;
```

Esempio:

```
throw new IllegalArgumentException();
```

Lancia una eccezione e restituisce il controllo ad un handler
Per il tipo di eccezione lanciata

Segnalare le eccezioni

- Nell'intestazione del metodo
- Esempio: quando di usa uno `Scanner` per leggere un file

```
String filename = . . .;
FileReader reader = new FileReader (filename);
Scanner in = new Scanner(reader);
```

Il costruttore `File` potrebbe in realtà lanciare una eccezione `FileNotFoundException` .

- Bisogna segnalarlo nell'intestazione del metodo

Segnalare le eccezioni

- Un metodo che chiama un altro metodo la cui esecuzione può lanciare un'eccezione, ha due possibili risposte:
  - gestire l'eccezione . . . dire al compilatore cosa fare
  - non gestire l'eccezione, ma dichiarare di poterla lanciare
    - Informare il compilatore che il metodo deve essere terminato a fronte di una eccezione
    - La clausola `throws` specifica che un metodo può lanciare una checked exception

```
public void read(String filename)
                          throws FileNotFoundException {
   File reader = new FileReader (filename);
   Scanner in = new Scanner(reader);
   . . .
}
```

Eccezioni Checked e Unchecked

- Un metodo può lanciare anche più di un tipo di eccezione:

```
public void read(String filename)
      throws IOException, ClassNotFoundException
```

- Le eccezioni definiscono la gerarchia ereditaria vista precedentemente:
  Se un metodo può lanciare sia `IOException` che `FileNotFoundException`, è sufficiente specificare `IOException`

Nota sintattica

```
accessSpecifier returnType
      methodName(parameterType parameterName, . . .)
            throws ExceptionClass, ExceptionClass, . . .
```

Esempio:

```
public void read(BufferedReader in) throws IOException
```

Specifica le eccezioni "checked" che il metodo può lanciare

Catturare le eccezioni

- Se si verifica un'eccezione, l'esecuzione del metodo in cui si è verificata interronpe la sua esecuzione, e di conseguenza tutto il programma
- È opportuno gestore ogni eccezione in modo tale che non sia causato l'arresto del programma
- Lo statement `try/catch` definisce un handler per un tipo di eccezione
- `try` block: racchiude gli statement che possono eventualmente causare una eccezione
- la clausola `catch`: contiene l'handler vero e proprio

Catturare le eccezioni

```
try {
   String filename = . . .;
   FileReader reader = new FileReader(filename);
   Scanner in = new Scanner(reader);
   String input = in.next();
   int value = Integer.parseInt(input);
   . . .
}
catch (IOException exception){
   exception.printStackTrace();
}
catch (NumberFormatException exception){
   System.out.println("Input was not a number");
}
```

IOException include FileNotFoundException e EOFException

Catturare le eccezioni

- Si esegue il blocco try
- Se non si verificano eccezioni, la parte **`catch`** viene ignorata
  - l'esecuzione procede con il codice che segue i blocchi catch
- Se si verifica una eccezione fra quelle "trattate" dal catch si salta alla clausola **`catch`** relativa
  - l'esecuzione procede con il codice che segue i blocchi catch
- Se si verifica un altro tipo di eccezione a cui i catch non si riferiscono, questa viene lanciata, ed il controllo viene passato all'exception handler

Catturare le eccezioni

- **`catch (IOException exception) block`**
  - **`exception`** è un riferimento all'oggetto Exception lanciato
  - catch può interrogare l'oggetto Exception
  - Si può chiamare nel blocco catch **`exception.printStackTrace()`**: stampa la catena di invocazioni di metodi che hanno provocato l'eccezione
    - Utile in fase del debugging del programma

Nota sintattica

```
try
{
   statement
   statement
   . . .
}
catch (ExceptionClass exceptionObject)
{
   statement
   statement
   . . .
}
catch (ExceptionClass exceptionObject)
{
   statement
   statement
   . . .
}
. . .
```

Nota sintattica

Esempio:

```
 try
{
   System.out.println("How old are you?");
   int age = in.nextInt();
   System.out.println("Next year, you'll be " + (age + 1));
}
catch (InputMismatchException exception)
{
   exception.printStackTrace();
}
```

Mettere a tacere le eccezioni

Esempio:

```
 try
{
   FileReader reader = new FileReader(filename)
      // il file potrebbe non esistere
      ...
}
catch (FileNotFoundException exception){}
```

- Se l'eccezione è lanciata, il gestore non fa nulla
- Da evitare perché le eccezioni sono state progettate per segnalare il problema
  - Nasconde una condizione d'errore che potrebbe essere seria

La clausola `finally`

- Abbiamo detto che una eccezione termina l'esecuzione del metodo corrente
- Può essere pericoloso, in quanto si salta codice essenziale

```
reader = new FileReader(filename);
Scanner in = new Scanner(reader);
readData(in);
reader.close();
// May never get here
```

La clausola `finally`

- E' importante eseguire lo statement **`reader.close()`** anche a fronte di una eccezione
- La clausola **`finally`** identifica una sezione di codice da eseguire "necessariamente" prima dell'abbandono di un metodo

La clausola `finally`

```
FileReader reader = new FileReader(filename);
try
{
   Scanner in = new Scanner(reader);
   readData(in);
}
finally
{
   reader.close(); // if an exception occurs,
                   // finally clause is also
                  // executed before exception is
                   //  passed to its handler
}
```

La clausola `finally`

- E' eseguita quando si esce da un blocco **`try`**:
  - Dopo l'ultimo statement (no eccezione!)
  - Dopo l'ultimo statement della clausola **`catch`**, se si verifica una eccezione
  - Quando una eccezione viene lanciata in quanto non trattata dall'handler

Nota sintattica

```
try{
   statement
   statement
   . . .
}
finally{
   statement
   statement
   . . .
}
```

Esempio:

```
FileReader reader = new FileReader(filename);
try {
   readData(reader);
}
finally {
   reader.close();
}
```

Clausola finally e try

- Attenzione: è preferibile evitare di mischiare **`catch`** e **`finally`** nello stesso blocco **`try`**

```
try {
      FileReader reader = new FileReader(filename);
      try {
            Scanner in = new Scanner(reader);
            readData(in);
      }
      finally {
            reader.close();
      }
}
catch (IOException exception) {
      //gestore dell'eccezione
}
```

Catturare eccezioni: Esempio

```
public class TestTry {
   public static void main(String[ ] arg)
                                  throws IOException {
      boolean ok=false;
       String fileName, s;
       Scanner scIn= new Scanner(System.in);
       System.out.println("Nome del file?");
       while(!ok) {
          try {
             fileName =sc.nextLine();
             Scanner sc = new Scanner(new File(fileName);
             ok=true;
             while((s=sc.nextLine())!=null)
                  System.out.println(s);
             }
           catch(FileNotFoundException e){
              System.out.print("File inesistente, nome?");
           }
      }
   }
}
```

Definire nuovi tipi di eccezione

- Se nessuna delle eccezioni di runtime ci sembra adeguata ad un caso specifico, il programmatore può progettarne una nuova
- Java consente di definire nuove sottoclassi di tipo exception, sia **`Exception`** che **`RuntimeException`**

```
if (amount > balance)
{
   throw new InsufficientFundsException(
     "withdrawal of " + amount + " exceeds balance of
        " + balance);
}
```

Definire nuovi tipi di eccezione

- Definiamo **`InsufficientFundsException`** come unchecked exception
  - Normalmente il programmatore si assicura di invocare prima **`getBalance`**
  - I nuovi tipi di eccezioni devono essere inseriti nella discendenza di **`Throwable`**, e in genere sono sottoclassi di **`RuntimeException`**.
- Due costruttori
  1. Default constructor
  2. Un costruttore con un parametro di tipo stringa che descrive il motivo dell'eccezione

Definire nuovi tipi di eccezione

```
public class InsufficientFundsException
      extends RuntimeException
{
   public InsufficientFundsException() {}

   public InsufficientFundsException(String message)
   {
      super(message);
   }
}
```