

Gestione delle eccezioni

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.
- Gli errori in Java sono definiti nella discendenza della classe `Throwable` nel package `java.lang`.
 - si possono usare le parole chiave di Java per la gestione degli errori solo su oggetti di tipo `Throwable`.

La Superclasse Throwable

- **Throwable** ha due sottoclassi dirette (sempre in `java.lang`)

- **Error**

- Errori fatali, dovuti a condizioni accidentali, non prevedibili (e quindi evitabili) dal programmatore
 - Esaurimento delle risorse di sistema necessarie alla JVM (OutOfMemoryError), incompatibilità di versioni, violazione di un'asserzione (AssertionError),
- I programmi non gestiscono questi errori

- **Exception**

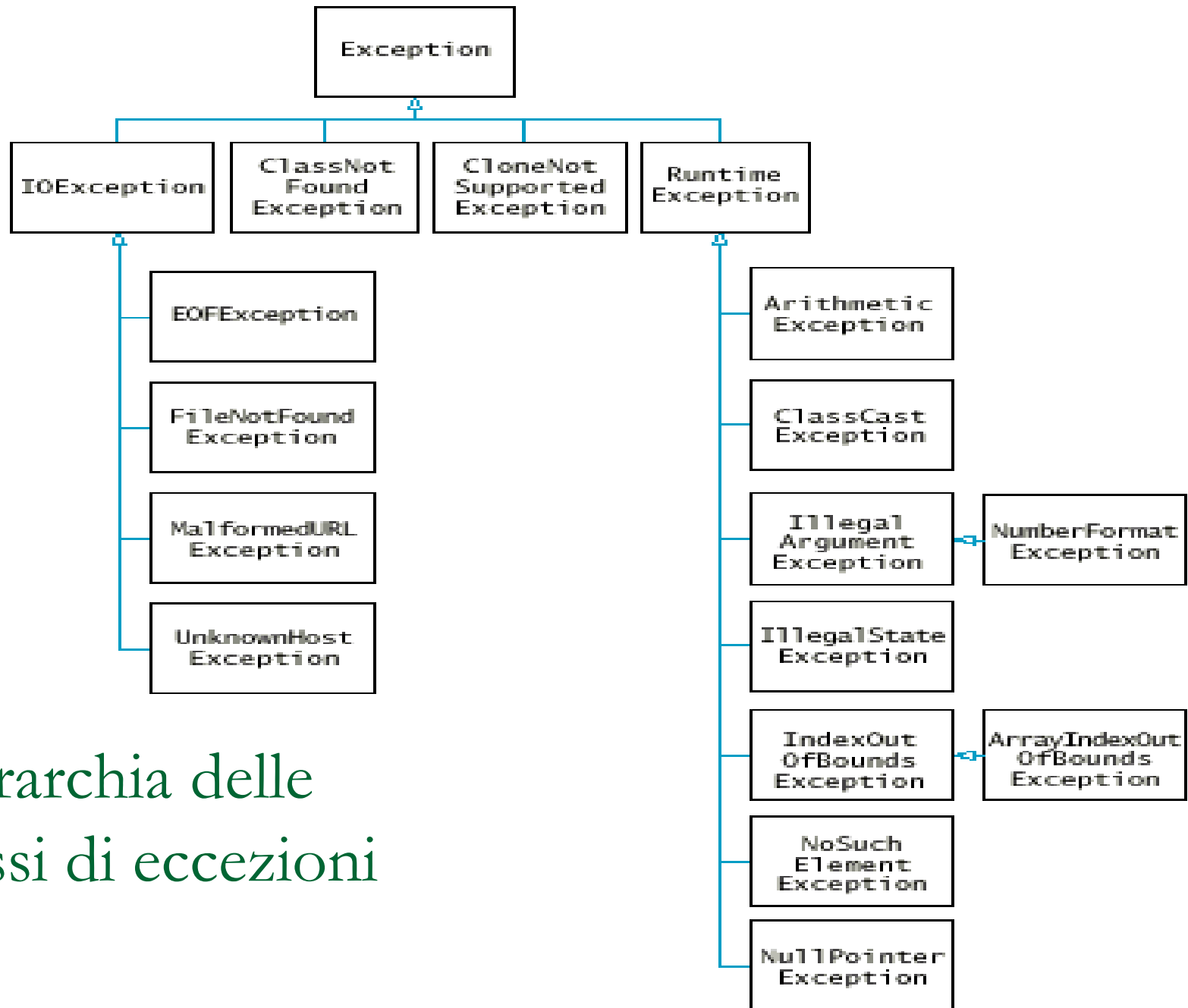
- Tutti gli errori che non rientrano in **Error**
 - I programmi possono gestire o no questi errori a seconda dei casi
-

Eccezioni

- Un'eccezione è un evento che interrompe la normale esecuzione del programma
- Se si verifica un'eccezione il controllo passa ad un gestore delle eccezioni
 - Il suo compito è di eseguire il codice previsto e quindi riprendere l'esecuzione normale oppure terminare con la segnalazione dell'errore

Eccezioni

- Java mette a disposizione varie classi di eccezioni, nei package
 - `java.lang`
 - `java.io`
- Tutte le classi che implementano eccezioni sono sottoclassi della classe `Exception`



Gerarchia delle
classi di eccezioni

Categorie di Eccezioni

- **eccezioni non controllate**

Non è obbligatorio per il programmatore gestire questo tipo di eccezioni

- in genere si usano per segnalare errori evitabili con un attenta programmazione

- **eccezioni controllate**

E' obbligatorio inserire un codice alternativo da eseguire oppure segnalare esplicitamente che il gestore delle eccezioni deve eseguire solo le operazioni di routine.

- in genere si usano in relazione ad errori caustati da eventi esterni (es., errore del disco, interruzione del collegamento di rete,..) e in tutte le situazioni che richiedono l'attenzione del programmatore (es., `CloneNotSupportedException`)

Esempi di eccezioni

- **eccezioni controllate**
 - **EOFException**: terminazione inaspettata del flusso di dati in ingresso
 - **FileNotFoundException**: file non trovato nel file system
- **eccezioni non controllate**
 - **NullPointerException**: uso di un riferimento null
 - **IndexOutOfBoundsException**: accesso ad elementi esterni ai limiti di un array

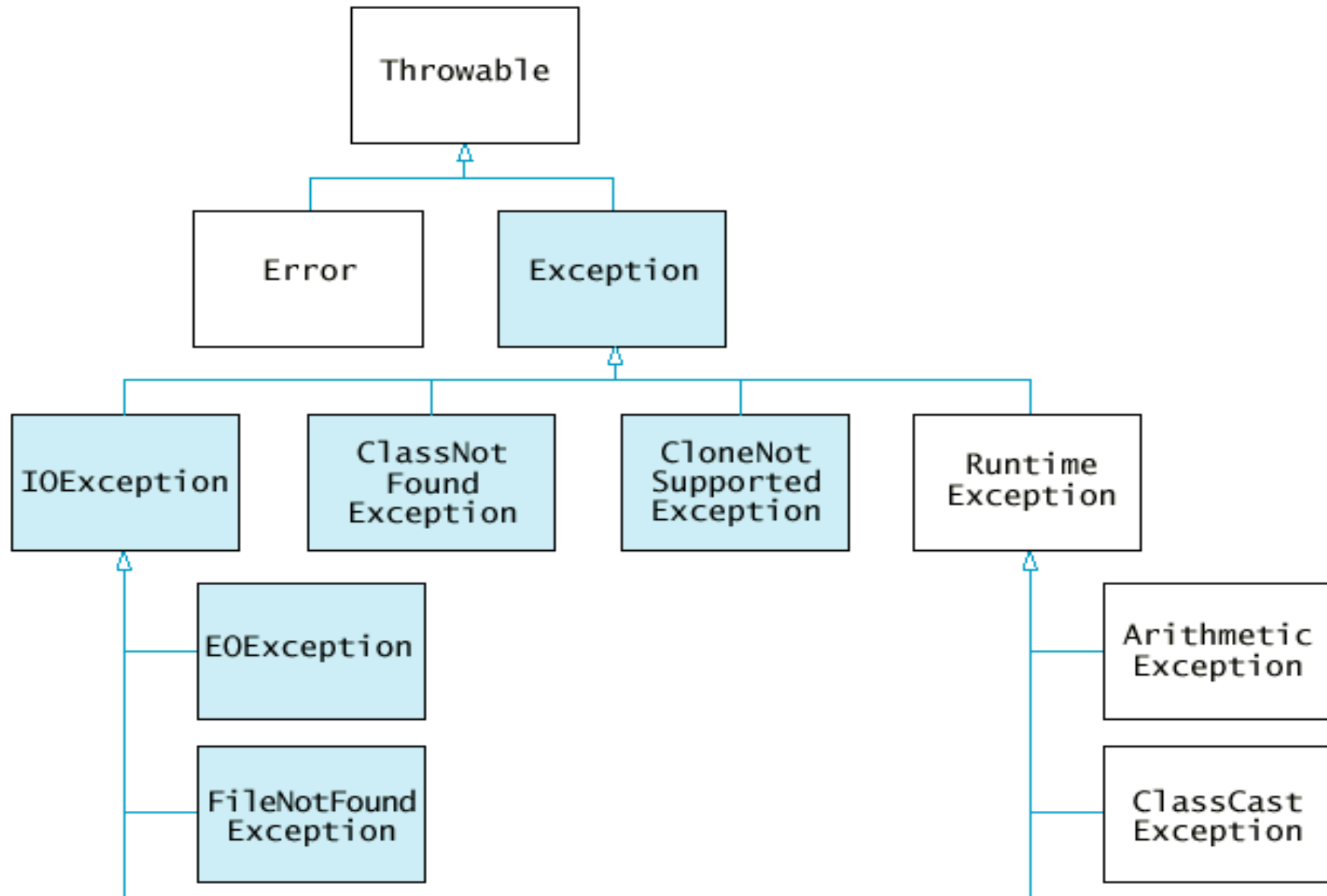
Eccezioni controllate

- 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 controllate e non controllate



Eccezioni

- Per lanciare un'eccezione, usiamo la parola chiave **throw** (lancia), seguita da un oggetto di tipo `Exception`
`throw exceptionObject;`
- Il metodo termina immediatamente e passa il controllo al **gestore delle eccezioni**
 - Le istruzioni successive non vengono eseguite

Lanciare eccezioni: esempio

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

La stringa in input al costruttore di `IllegalArgumentException` rappresenta il messaggio d'errore che viene visualizzato quando si verifica l'eccezione

Segnalare eccezioni controllate

- `Object.clone()` può lanciare una `CloneNotSupportedException`
- Un metodo che invoca `clone()` può
 - ❑ **gestire l'eccezione**, cioè dire al compilatore cosa fare
 - ❑ **non gestire l'eccezione**, ma dichiarare di poterla lanciare
 - In tal caso, se l'eccezione viene lanciata, il programma termina visualizzando un messaggio di errore

Segnalare eccezioni

- Per segnalare le eccezioni controllate che il metodo può lanciare usiamo la parola chiave `throws`
- Esempio:

```
public class Customer implements Cloneable
{
    ...
    public Object clone() throws CloneNotSupportedException
    {
        Customer cloned = (Customer) super.clone();
        cloned.account = (BankAccount) account.clone();
        return cloned;
    }

    private String name;
    private BankAccount account;
}
```


Segnalare eccezioni

- Qualunque metodo che chiama `x.clone()` (dove `x` è un oggetto di tipo `Customer`) deve decidere se gestire l'eccezione o dichiarare di poterla lanciare

```
public class ArchivioClienti
{
    public void calcola(int i) throws
                                CloneNotSupportedException
    {
        .....
        Customer c = clienti.get(i).clone();
        .....
    }
    .....
}
```

Segnalare eccezioni

- Un metodo può lanciare più eccezioni controllate, di tipo diverso

```
public void calcola(int i)  
    throws CloneNotSupportedException,  
           ClassNotFoundException
```

Usare le Eccezioni di Runtime

- Le eccezioni di runtime (**RuntimeException**) possono essere utilizzate per segnalare problemi dovuti a dati al di fuori del dominio “corretto”.

- Esempi:

- Un metodo che preleva soldi da un conto corrente non può prelevare una quantità maggiore del saldo
- Un metodo che effettua una divisione non può dividere un numero per zero

Progettare Nuove Eccezioni

- Se nessuna delle eccezioni ci sembra adeguata al nostro caso, possiamo progettarne una nuova.
- I nuovi tipi di eccezioni devono essere inseriti nella discendenza di `Throwable`
- Per definire una nuova eccezione
 - a controllo non obbligatorio (non controllata) si estende una classe nella discendenza di `RuntimeException` (in viene estesa direttamente questa classe)
 - a controllo obbligatorio (controllata) si estende una qualsiasi altra classe nella discendenza di `Exception` (in genere direttamente `Exception`)

Progettare Nuove Eccezioni

Introduciamo un nuovo tipo di eccezione per controllare che il denominatore sia diverso da zero, prima di eseguire una divisione:

```
public class DivisionePerZeroException extends
    RuntimeException{

    public DivisionePerZeroException() {
        super("Divisione per zero!");
    }

    public DivisionePerZeroException(String msg) {
        super(msg);
    }
}
```

Usare Nuove Eccezioni

```
public class Divisione {  
    public Divisione(int n, int d) {  
        num=n;  
        den=d;  
    }  
    public double dividi() {  
        if (den==0)  
            throw new DivisionePerZeroException();  
        return num/den;  
    }  
    private int num;  
    private int den;  
}
```

Usare Nuove Eccezioni: Esempio

```
public class Test {  
    public static void main(String[] args) {  
        double res;  
        Scanner in = new Scanner(System.in);  
  
        System.out.print("Inserisci il numeratore:");  
        int n= in.nextInt();  
        System.out.print("Inserisci il denominatore:");  
        int d= in.nextInt();  
  
        Divisione div = new Divisione(n,d);  
        res = div.dividi();  
    }  
}
```

Usare Nuove Eccezioni: Esempio

Inserisci il numeratore: 5

Inserisci il denominatore: 0

DivisionePerZeroException: Divisione per zero!

at Divisione.dividi(Divisione.java:12)

at divisioneperzero.Test.main(Test.java:22)

Exception in thread "main"

- Il **main** invoca il metodo **dividi** della classe **Divisione** alla linea 22
- Il metodo **dividi** genera una eccezione alla linea 12

Catturare eccezioni

- Le eccezioni devono essere gestite per evitare l'arresto del programma
- Per installare un gestore si usa l'enunciato `try`, seguito da tante clausole `catch` quante sono le eccezioni che si vogliono gestire

Catturare eccezioni

```
try
{
    istruzione
    istruzione
    . . .
}
catch (ClasseEccezione oggettoEccezione)
{
    istruzione
    istruzione
    . . .
}
catch (ClasseEccezione oggettoEccezione)
{
    istruzione
    istruzione
    . . .
}
. . .
```

Catturare eccezioni

- Vengono eseguite le istruzioni all'interno del blocco `try`
- Se nessuna eccezione viene lanciata, le clausole `catch` sono ignorate
- Se viene lanciata un'eccezione viene eseguita la corrispondente clausola `catch`

Catturare Eccezioni: Esempio

```
public class Test {  
    public static void main(String[] args){  
        double res;  
        Scanner in = new Scanner(System.in);  
  
        System.out.print("Inserisci il numeratore:");  
        int n= in.nextInt();  
  
        System.out.print("Inserisci il denominatore:");  
        int d= in.nextInt();  
    }  
}
```

Catturare Eccezioni: Esempio

```
try
{
    Divisione div = new Divisione(n,d);
    res = div.dividi();
    System.out.print(res);
}

catch (DivisionePerZeroException exception)
{
    System.out.println(exception);
}
}
```

Catturare eccezioni

- Cosa fa l'istruzione

`System.out.println(exception) ?`

- Invoca il metodo `toString()` della classe `DivisioneperZeroException`

- Ereditato dalla classe `RuntimeException`

- Restituisce una stringa che descrive l'oggetto `exception` costituita da

- Il nome della classe a cui l'oggetto appartiene seguito da ":" e dal messaggio di errore associato all'oggetto

Catturare Eccezioni: Esempio

Inserisci il numeratore:5

Inserisci il denominatore:0

DivisionePerZeroException: Divisione per zero!

- **DivisionePerZeroException**

- è la classe a cui l'oggetto **exception** appartiene

- **Divisione per zero!**

- È il messaggio di errore associato all'oggetto **exception** (dal costruttore)

Catturare eccezioni

- Per avere un messaggio di errore che stampa lo stack delle chiamate ai metodi in cui si è verificata l'eccezione usiamo il metodo `printStackTrace()`

```
catch (DivisionePerZeroException exception)
{
    exception.printStackTrace();
}
```

- Output:

```
Inserisci il numeratore: 5
Inserisci il denominatore: 0
DivisionePerZeroException: Divisione per zero!
at Divisione.dividi(Divisione.java:12)
at divisioneperzero.Test.main(Test.java:22)
```

Catturare eccezioni

- Scriviamo un programma che chiede all'utente il nome di un file
- Se il file esiste, il suo contenuto viene stampato a video
- Se il file non esiste viene generata un'eccezione
- Il gestore delle eccezioni avvisa l'utente del problema e gli chiede un nuovo file

Catturare eccezioni: Esempio

```
import java.io.*;
public class TestTry {

    public static void main(String[ ] arg)
        throws IOException {

        Scanner in = new Scanner(System.in);

        boolean ok=false;

        String s;

        System.out.println("Nome del file?");
```

Catturare eccezioni: Esempio

```
while(!ok) {  
    try {  
        s=in.next();  
        FileReader fr=new FileReader(s);  
        in=new Scanner(fr);  
        ok=true;  
        while( (s=in.nextLine()) !=null)  
            System.out.println(s);  
    }  
    catch(FileNotFoundException e) {  
        System.out.println("File  
            inesistente, nome?");  
    }  
}  
}
```

La clausola `finally`

- Il lancio di un'eccezione arresta il metodo corrente
- A volte vogliamo eseguire altre istruzioni prima dell'arresto
- La clausola `finally` viene usata per indicare un'istruzione che va eseguita sempre
 - Ad, esempio, se stiamo leggendo un file e si verifica un'eccezione, vogliamo comunque chiudere il file

La clausola finally

```
try
{
    istruzione
    istruzione
    ...
}
finally
{
    istruzione
    istruzione
    ...
}
```

La clausola `finally`

- Viene eseguita al termine del blocco `try`
- Viene comunque eseguita se un'istruzione del blocco `try` lancia un'eccezione
- Può anche essere combinata con clausole `catch`

La clausola finally

```
FileReader reader =  
    new FileReader(filename);  
try  
{  
    Scanner in = new Scanner(reader);  
    readData(in);  
    //metodo di lettura dati  
}  
  
finally  
{  
    reader.close();  
}
```

File BadDataException.java

```
public class BadDataException extends
    RuntimeException{

    public BadDataException() {}

    public BadDataException(String msg) {
        super(msg) ;
    }
}
```


File DataSetReader.java

```
01: import java.io.FileReader;
02: import java.io.IOException;
03: import java.util.Scanner;
04:
05: /**
06:  Reads a data set from a file. The file must have the format:
07:      numberOfValues
08:      value1
09:      value2
10:      . . .
11: */
12: public class DataSetReader
13: {
```

File DataSetReader.java

```
14:    /**
15:        Reads a data set.
16:        @param filename the name of the file holding the data
17:        @return the data in the file
18:    */
19:    public double[] readFile(String filename)
20:        throws IOException, BadDataException
21:    {
22:        FileReader reader = new FileReader(filename);
23:        try
24:        {
25:            Scanner in = new Scanner(reader);
26:            readData(in);
27:        }
28:        finally
29:        {
30:            reader.close();
31:        }
```

File DataSetReader.java

```
32:         return data;
33:     }
34:
35:     /**
36:      Reads all data.
37:      @param in the scanner that scans the data
38:     */
39:     private void readData(Scanner in) throws BadDataException
40:     {
41:         if (!in.hasNextInt())
42:             throw new BadDataException("Length expected");
43:         int numberOfValues = in.nextInt();
44:         data = new double[numberOfValues];
45:
46:         for (int i = 0; i < numberOfValues; i++)
47:             readValue(in, i);
```

File DataSetReader.java

```
48:
49:     if (in.hasNext())
50:         throw new BadDataException("End of file expected");
51:     }
52:
53:     /**
54:      Reads one data value.
55:      @param in the scanner that scans the data
56:      @param i the position of the value to read
57:     */
58:     private void readValue(Scanner in, int i)
59:         throws BadDataException
60:     {
```

File DataSetReader.java

```
60:         if (!in.hasNextDouble())
61:             throw new BadDataException("Data value expected");
62:         data[i] = in.nextDouble();
63:     }
64:
65:     private double[] data;
66: }
```

File DataSetTester.java

```
01: import java.io.FileNotFoundException;
02: import java.io.IOException;
03: import java.util.Scanner;
04:
05: public class DataSetTester
06: {
07:     public static void main(String[] args)
08:     {
09:         Scanner in = new Scanner(System.in);
10:         DataSetReader reader = new DataSetReader();
11:
12:         boolean done = false;
13:         while (!done)
14:         {
15:             try
16:             {
```

File DataSetTester.java

```
17:         System.out.println("Please enter the file name: ");
18:         String filename = in.next();
19:
20:         double[] data = reader.readFile(filename);
21:         double sum = 0;
22:         for (double d : data) sum = sum + d;
23:         System.out.println("The sum is " + sum);
24:         done = true;
25:     }
26:     catch (FileNotFoundException exception)
27:     {
28:         System.out.println("File not found.");
29:     }
30:     catch (BadDataException exception)
31:     {
32:         System.out.println
            ("Bad data: " + exception.getMessage());
```

File DataSetTester.java

```
33:         }
34:         catch (IOException exception)
35:         {
36:             exception.printStackTrace();
37:         }
38:     }
39: }
40: }
```

Vantaggi nell'uso delle eccezioni

- Separazione tra codice per il trattamento degli errori e codice “regolare”
- Propagazione degli errori attraverso le chiamate ricorsive nello stack
- Raggruppamento e differenziazione dei tipi di errore

Separazione codice Errore/Regolare (1)

- Le eccezioni forniscono un mezzo per separare i dettagli di cosa fare quando qualcosa di straordinario accade dalla logica principale del programma.
- Esempio: pseudo-codice lettura di un intero file nella memoria principale
 - readFile {
 open the file;
 determine its size;
 allocate that much memory;
 read the file into memory;
 close the file;
}

Separazione codice Errore/Regolare (2)

- Il metodo `readFile` sembra a posto ma ignora errori potenziali:
Cosa accade se..
 - ❑ il file non può essere aperto
 - ❑ la lunghezza del file non può essere determinata
 - ❑ non può essere allocata memoria sufficiente
 - ❑ l'operazione di lettura ha un fallimento
 - ❑ il file non può essere chiuso
- Per trattare questi casi, il metodo `readFile` dovrebbe contenere codice aggiuntivo per l'individuazione, segnalazione e trattamento dell'errore.

Separazione codice Errore/Regolare (3)

- Ad esempio, il codice di readFile potrebbe essere modificato in questo modo

```
□ readFile {  
    initialize errorCode = 0;  
    open the file;  
    if (theFileIsOpen) {  
        determine the length of the file;  
        if (gotTheFileLength) {  
            allocate that much memory;  
            if (gotEnoughMemory) {  
                read the file into memory;  
                if (readFailed) { errorCode = -1; }  
            } else { errorCode = -2; }  
        } else { errorCode = -3; }  
        close the file;  
        if (theFileDidntClose) { errorCode += -5; }  
    } else { errorCode = -4; }  
    return errorCode;  
}
```

Separazione codice Errore/Regolare (4)

- Le eccezioni consentono di scrivere il flusso principale del codice in un punto e occuparsi dei casi di errore altrove.
- **Osservazione:**
le eccezioni non ci permettono di risparmiare il lavoro per l'individuazione, segnalazione e trattamento dell'errore ma ci aiutano a organizzare il lavoro in maniera più efficiente

Separazione codice Errore/Regolare (4)

- Versione readFile con eccezioni al posto delle tecniche tradizionali di gestione dell'errore:

```
□ readFile {  
    try {  
        open the file;  
        determine its size;  
        allocate that much memory;  
        read the file into memory;  
        close the file;  
    } catch (fileOpenFailed) {  
        doSomething;  
    } catch (sizeDeterminationFailed) {  
        doSomething;  
    } catch (memoryAllocationFailed) {  
        doSomething;  
    } catch (readFailed) {  
        doSomething;  
    } catch (fileCloseFailed) {  
        doSomething;  
    }  
}
```

Propagazione errori nel call-stack (1)

- Si supponga:

```
main { .....
```

```
    call method1;
```

```
    .....}
```

```
method1 { .....
```

```
    call readFile;
```

```
    .....}
```

- Supponiamo che gli errori in `readFile` possano essere gestiti solo nel metodo `main`

Propagazione errori nel call-stack (2)

- Soluzione tradizionale

```
main { .....  
    errorCodeType error;  
    error = call method1;  
    if (error) doErrorProcessing;  
    else proceed;  
    ....}
```

```
errorCodeType method1 { .....  
    errorCodeType error;  
    error = call readFile;  
    if (error) return error;  
    else proceed;  
    .....}
```


Propagazione errori nel call-stack (3)

- Con le eccezioni:

```
main { .....  
    try { call method1; }  
    catch (exception e) {  
        doErrorProcessing;  
    }  
    ....}
```

```
method1 throws exception { .....  
    call readFile;  
    .....}
```

- Nota: il JRE ricerca all'indietro nelle chiamate sospese nel call-stack finché non trova un metodo che gestisce l'eccezione

Raggruppamento e differenziazione

- Raggruppamento è una conseguenza naturale della gerarchia
 - FileNotFoundException è un IOException
- Possiamo creare gruppi di eccezioni e trattare le eccezioni per il tipo generale di eccezione (IOException)
- Oppure possiamo usare il tipo specifico dell'eccezione e trattare l'eccezione in maniera specifica (FileNotFoundException)