



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, ...



Gestione Imprevisti

- Un buon programma gestisce gli imprevisti così come gestisce le cose previste
- Ad esempio `int i = Integer.parseInt(br.next());`
`parseInt` non può correggere l'errore, neanche identificare la sorgente dell'errore.
- L'argomento è stato trasmesso dal metodo invocante
- ...`parseInt` può solo comunicare all'invocante che si è presentato un errore



Gestione Imprevisti

- Che fare quando si presenta una situazione non prevista nell'esecuzione di un programma?
 1. Segnalare la condizione di errore attraverso la restituzione di valori atipici

In JAVA la gestione degli errori può essere fatta usando il meccanismo delle **eccezioni**, che sono oggetti che possono essere creati e lanciati(**throw**) in determinate condizioni, e che possono essere catturati (**catch**) dal codice scritto appositamente per la loro gestione

1. Le eccezioni non devono poter essere trascurate
2. Le eccezioni devono poter essere gestite da uno gestore competente, non semplicemente dal chiamante del metodo che fallisce



Eccezioni

- Informalmente, un'eccezione è un'anomalia, di cui sia possibile effettuare il recupero, occorsa durante l'esecuzione di un programma.
- Formalmente, un'eccezione è una violazione di vincoli semantici o di risorsa.
- Esempi tipici di eccezioni sono:
 - Divisione di un numero per zero
 - Indirizzamento oltre la dimensione di un vettore
 - Errore di tipo in accesso a fonti dati esterne



Esempi

- Zero.java può causare un'eccezione

```
java.lang.ArithmeticException: / by zero
    at Zero.calcolaQuoziente(Zero.java:27)
    at Zero.main(Zero.java:21)
Exception in thread "main" Process Exit...
```

- BasicArray_eccezione.java causa l'eccezione

```
java.lang.ArrayIndexOutOfBoundsException
    at BasicArray_eccezione.main(BasicArray_eccezione.java:30)
Exception in thread "main" Process Exit...
```



Esempi

- Postfissa.java può causare eccezioni

```
java.util.EmptyStackException
    at java.util.Stack.peek(Stack.java:82)
    at java.util.Stack.pop(Stack.java:64)
    at Postfissa.elabora(Postfissa.java:37)
    at Postfissa.main(Postfissa.java:21)
```

Exception in thread "main" Process

```
java.lang.NumberFormatException: 1=
    at java.lang.Integer.parseInt(Integer.java:423)
    at java.lang.Integer.<init>(Integer.java:549)
    at Postfissa.elabora(Postfissa.java:42)
    at Postfissa.main(Postfissa.java:21)
```

Exception in thread "main" Process Exit...



Eccezioni

- 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**
 - Il suo compito è quello di uscire dal frammento di codice che ha generato l'eccezione e decidere cosa fare



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**
 - solo sugli oggetti di questa classe si possono usare le parole chiave di java per la gestione degli errori.



La Superclasse Throwable

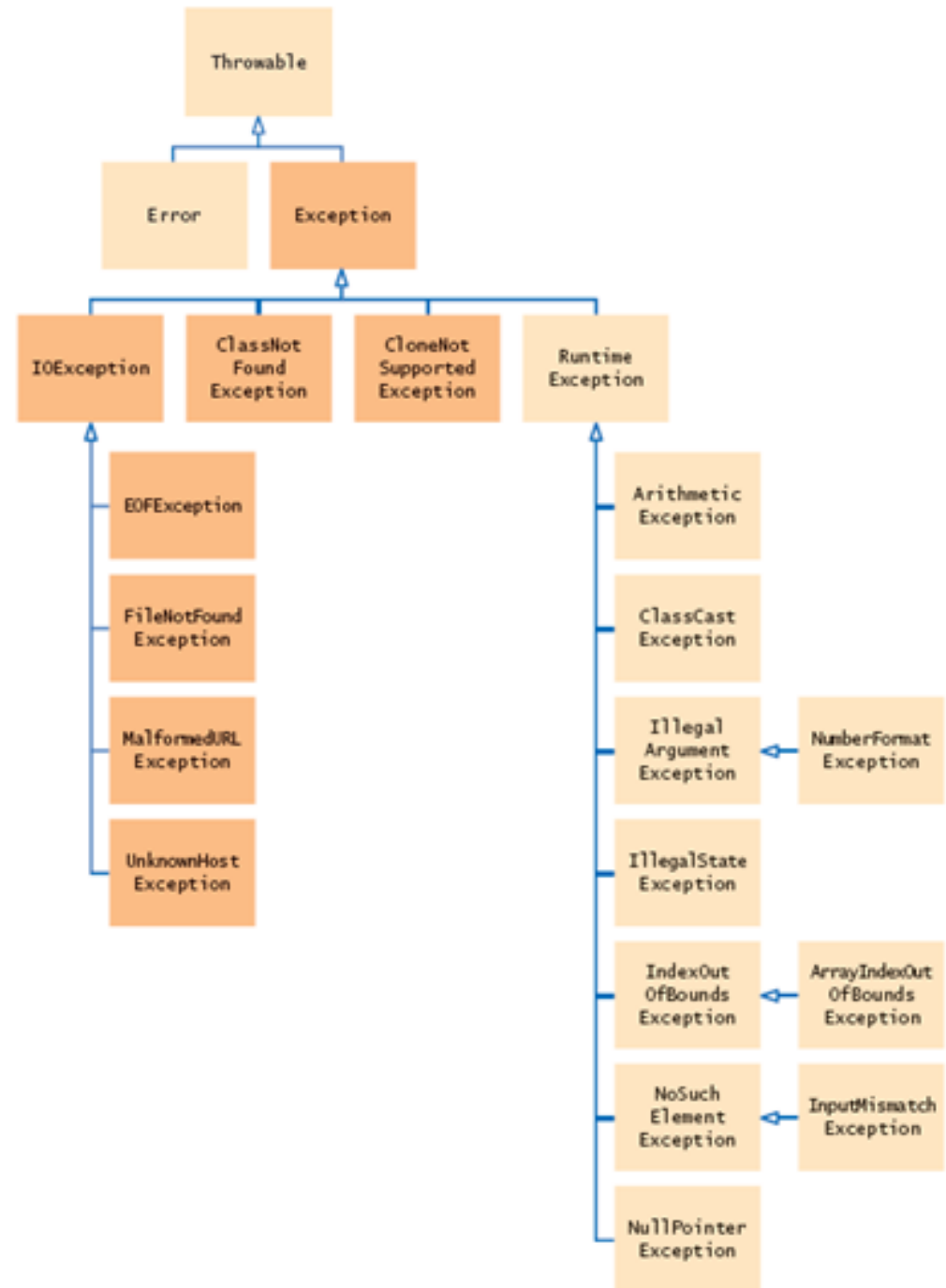
- La superclasse **Throwable** ha due sottoclassi dirette, sempre in **java.lang**
- **Error**
 - Errori fatali, dovuti a condizioni accidentali
 - Esaurimento delle risorse di sistema necessarie alla JVM (**OutOfMemoryError**), incompatibilità di versioni, violazione di un'asserzione (**AssertionError**),
 - In genere 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

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

Gerarchia delle classi di eccezioni





Categorie di Eccezioni

- **eccezioni non controllate**
 - dovute a circostanze che il programmatore **può evitare**, correggendo il programma
- **eccezioni controllate**
 - dovute a circostanze esterne che il programmatore **non può evitare**
 - il compilatore vuole sapere cosa fare nel caso si verifichi l'eccezione



Categorie di Eccezioni

- Esempio di **eccezione controllata**
 - **EOFException**: terminazione inaspettata del flusso di dati in ingresso
 - Può essere provocata da eventi esterni
 - errore del disco
 - interruzione del collegamento di rete
 - Il gestore dell'eccezione si occupa del problema



Tipi di eccezioni

- Esempi di **eccezione non controllata**
 - **NullPointerException**: uso di un riferimento **null**
 - **IndexOutOfBoundsException**: accesso ad elementi esterni ai limiti di un array
- Non è obbligatorio scrivere un codice per gestire questo tipo di eccezione
 - Il programmatore può prevenire queste anomalie, correggendo il codice



Eccezioni controllate

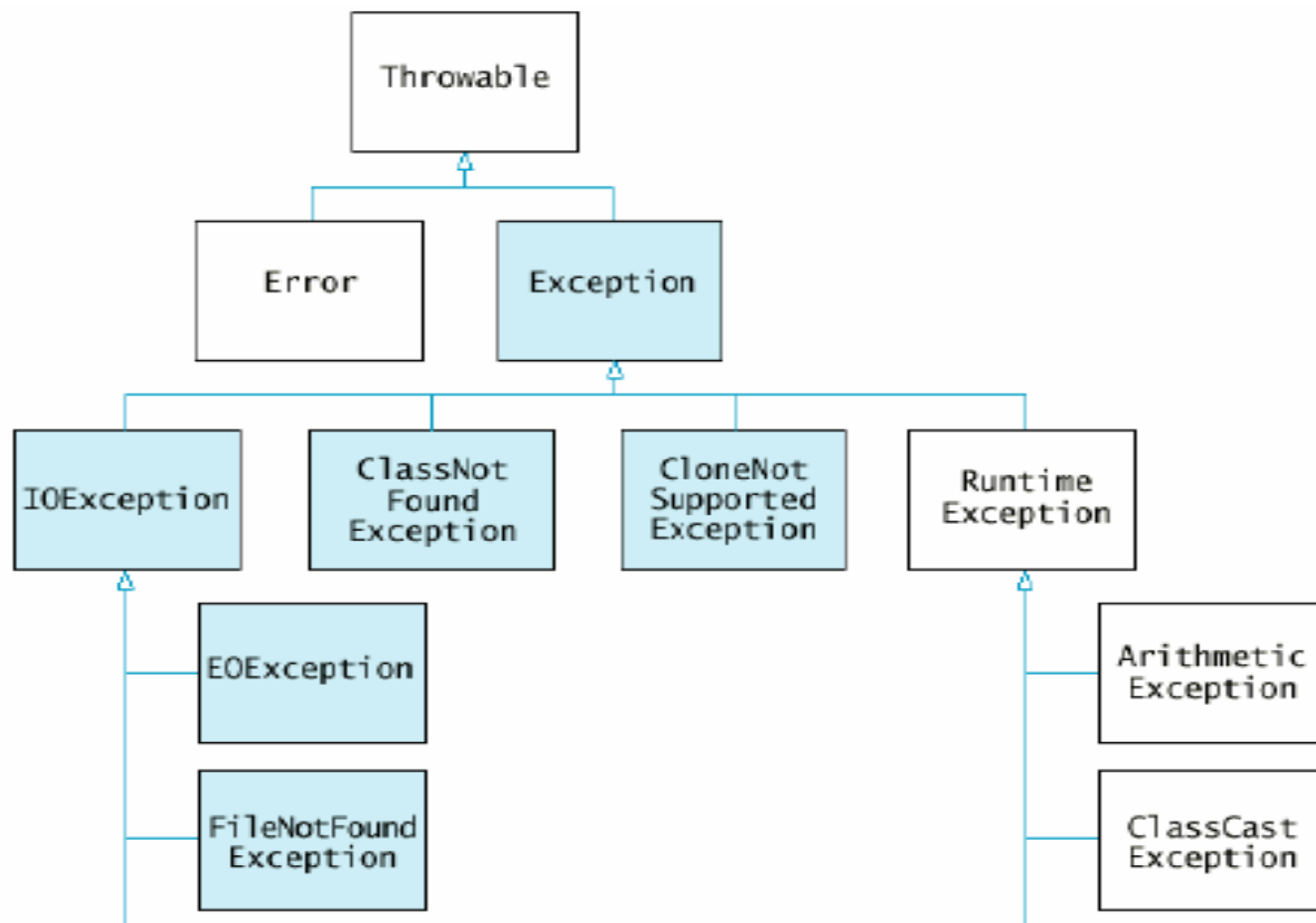
- Tutte le sottoclassi di **IOException**
 - **EOFException**
 - **FileNotFoundException**
 - **MalformedURLException**
 - **UnknownHostException**
- **ClassNotFoundException**
- **CloneNotSupportedException**



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 eccezione

throw exceptionObject;

Syntax **throw** *exceptionObject*;

Example

A new exception object is constructed, then thrown.

```
if (amount > balance)
{
    throw new IllegalArgumentException("Amount exceeds balance");
}
balance = balance - amount;
```

Most exception objects can be constructed with an error message.

This line is not executed when the exception is thrown.



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 (*a meno che non venga gestita dal chiamante del metodo*)



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 se 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 ad input errati.
- 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



Catturare eccezioni

- Ogni eccezione deve essere gestita, altrimenti causa l'arresto del programma
- Per installare un gestore si usa l'enunciato `try`, seguito da tante clausole `catch` quante sono le eccezioni da gestire



Catturare eccezioni

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

Catturare eccezioni

Syntax

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

Example

When an `IOException` is thrown, execution resumes here.

Additional catch clauses can appear here.

```
try
{
    Scanner in = new Scanner(new File("input.txt"));
    String input = in.next();
    process(input);
}
catch (IOException exception)
{
    System.out.println("Could not open input file");
}
```

This constructor can throw a `FileNotFoundException`.

This is the exception that was thrown.

A `FileNotFoundException` is a special case of an `IOException`.



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**
 - E' la classe a cui l'oggetto **exception** appartiene
- **Divisione per zero!**
 - E' 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

Syntax

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

Example

This code may
throw exceptions.

This code is
always executed,
even if an exception occurs.

This variable must be declared outside the try block
so that the finally clause can access it.

```
PrintWriter out = new PrintWriter(filename);
try
{
    writeData(out);
}
finally
{
    out.close();
}
```



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



La clausola **finally**

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



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`, e in genere sono sottoclassi di `RuntimeException`.
- Un tipo di eccezione che sia sottoclasse di `RuntimeException` sarà a controllo **non obbligatorio**.
- Per definire una nuova eccezione di tipo controllato allora deve essere sottoclasse di `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



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: {
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);
```



File DataSetReader.java

```
23:         try
24:         {
25:             Scanner in = new Scanner(reader);
26:             readData(in);
27:         }
28:         finally
29:         {
30:             reader.close();
31:         }
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];
```




File DataSetReader.java

```
45:
46:     for (int i = 0; i < numberOfValues; i++)
47:         readValue(in, i);
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) throws
BadDataException
59: {
60:     if (!in.hasNextDouble())
61:         throw new BadDataException("Data value expected");
62:     data[i] = in.nextDouble();
63: }
64:
65: private double[] data;
66: }
```



File BadDataException.java

```
public class BadDataException extends
    Exception{

    public BadDataException() {}

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

File DataAnalyzer.java

```
01: import java.io.FileNotFoundException;
02: import java.io.IOException;
03: import java.util.Scanner;
04:
05: /**
06:     This program reads a file containing numbers and analyzes its contents.
07:     If the file doesn't exist or contains strings that are not numbers, an
08:     error message is displayed.
09: */
10: public class DataAnalyzer
11: {
12:     public static void main(String[] args)
13:     {
14:         Scanner in = new Scanner(System.in);
15:         DataSetReader reader = new DataSetReader();
16:
17:         boolean done = false;
18:         while (!done)
19:         {
20:             try
21:             {
22:                 System.out.println("Please enter the file name: ");
23:                 String filename = in.next();
```

File DataAnalyzer.java

```
24:
25:         double[] data = reader.readFile(filename);
26:         double sum = 0;
27:         for (double d : data) sum = sum + d;
28:         System.out.println("The sum is " + sum);
29:         done = true;
30:     }
31:     catch (FileNotFoundException exception)
32:     {
33:         System.out.println("File not found.");
34:     }
35:     catch (BadDataException exception)
36:     {
37:         System.out.println("Bad data: " + exception.getMessage());
38:     }
39:     catch (IOException exception)
40:     {
41:         exception.printStackTrace();
42:     }
43: }
44: }
45: }
```



La gestione di eccezioni: compile-time

- Per la gestione delle eccezioni, il compilatore Java effettua due controlli:
 1. Per ogni eccezione lanciabile dal codice deve esistere un handler menzionante la classe (o una superclasse) di tale eccezione.
 2. Non devono esistere handler per eccezioni non lanciabili dal codice.



La gestione di eccezioni: compile-time

- Tali controlli non si applicano alle **eccezioni non controllate** (da qui il nome), poiché quasi ogni istruzione potenzialmente può provocare un'eccezione e, a giudizio dei progettisti di Java, il doverle gestire con dei try-catch sarebbe stato di grande fastidio per i programmatori.
- Ad esempio, ogni divisione può generare un' **ArithmeticException**, ma dover definire ogni istruzione che utilizza tale operatore in un blocco di try sarebbe oltre modo noioso.



La gestione di eccezioni: run-time

- Al lancio di un'eccezione, il controllo viene trasferito dal codice anomalo alla clausola catch che gestisce l'eccezione
- Il trasferimento di controllo causa la brutale terminazione di espressioni o istruzioni: l'esecuzione continua quindi nel blocco del catch, ed il codice causante l'eccezione non potrà più continuare la sua esecuzione.




La gestione di eccezioni: run-time

- La difficoltà principale in questa gestione è nel determinare il giusto handler.
- La discriminazione è complicata da tre possibilità:
 1. Più catch dopo un blocco di try
 2. Catch referenzianti superclassi dell'eccezione
 3. Blocchi di try innestati



Determinazione dell' handler

- La determinazione della giusta clausola catch è effettuata confrontando la classe dell'oggetto lanciato col tipo dichiarato come parametro del catch.
- Il matching si ha se il tipo dei parametri del catch è la classe (o una superclasse) dell'eccezione.
- In presenza di più match viene selezionata la prima clausola.
- Non viene data alcuna priorità ad una corrispondenza esatta rispetto ad una corrispondenza che richiede l'applicazione di conversioni standard



```
public class Conta{
    public static int counter(int[] n) throws Exception {
        int ris = 0;
        for(int i=0; i < n.length; i++) {
            try { ris += n[i] / (n[i] % 3); }
            catch(Exception e) {
                if (ris <= 5) ris++;
                else throw new Exception(String.valueOf(ris));
            }
        }
        return ris;
    }
}

public class Boot {
    public static void main(String[] args) {
        int ris = 0, a[] = { 2, 3, 4, 5, 6, 7};
        try { ris = Conta.counter(a); }
        catch(Exception e) {
            System.out.println("errore: " + e.getMessage());
        }
        System.out.println("ris: " + ris);
    }
}
```