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, …

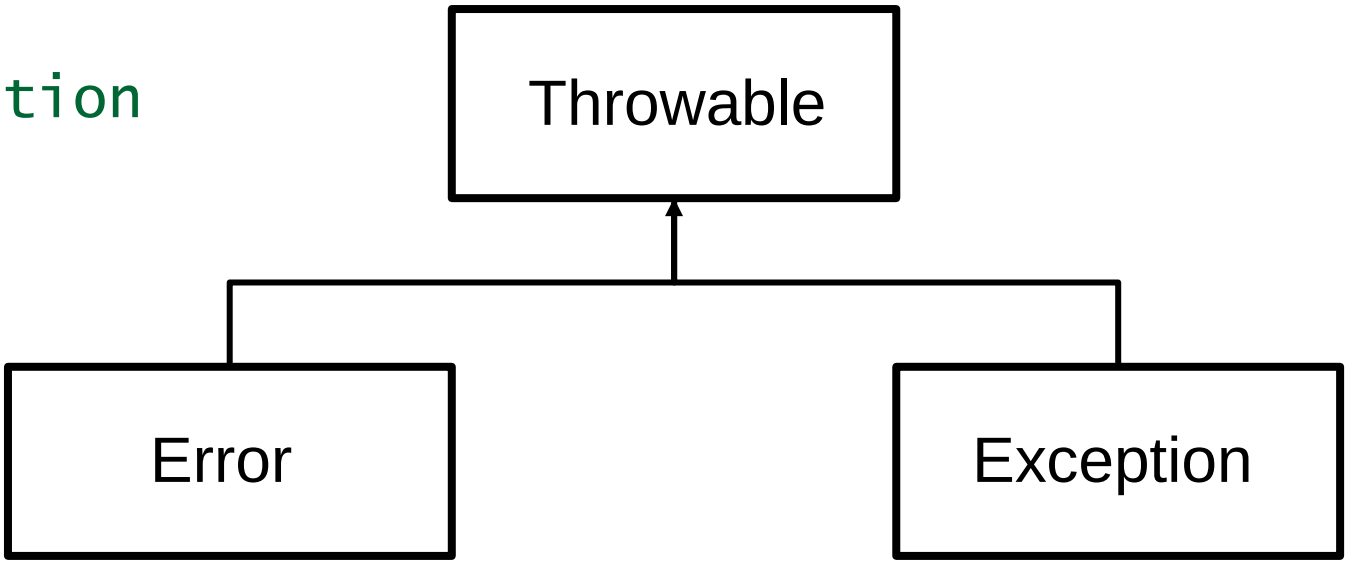
Condizioni di Errore in java

- 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 classe Throwable (java.lang)

- `Throwable` è la superclasse di tutti gli errori e le eccezioni in Java
  - definisce il caso base di ogni cosa che può essere lanciata
- Ha due sottoclassi dirette (sempre in `java.lang`)
  - `Error`
  - `Exception`



Classe Error

- Utilizzata per indicare una situazione anomala in un'esecuzione, fuori dal controllo dell'applicazione stessa
  - Ad es.
    - esaurimento delle risorse di sistema necessarie alla JVM (OutOfMemoryError)
    - incompatibilità di versioni
    - violazione di un'asserzione (AssertionError)
    - ……
- Indica un problema serio, dovuto a condizioni accidentali non prevedibili (e quindi evitabili dal programmatore)
- Questi errori non dovrebbero essere gestiti dai programmi
  - situazione compromessa al punto che un programma non sarebbe in grado di gestirli in maniera affidabile

Classe Exception

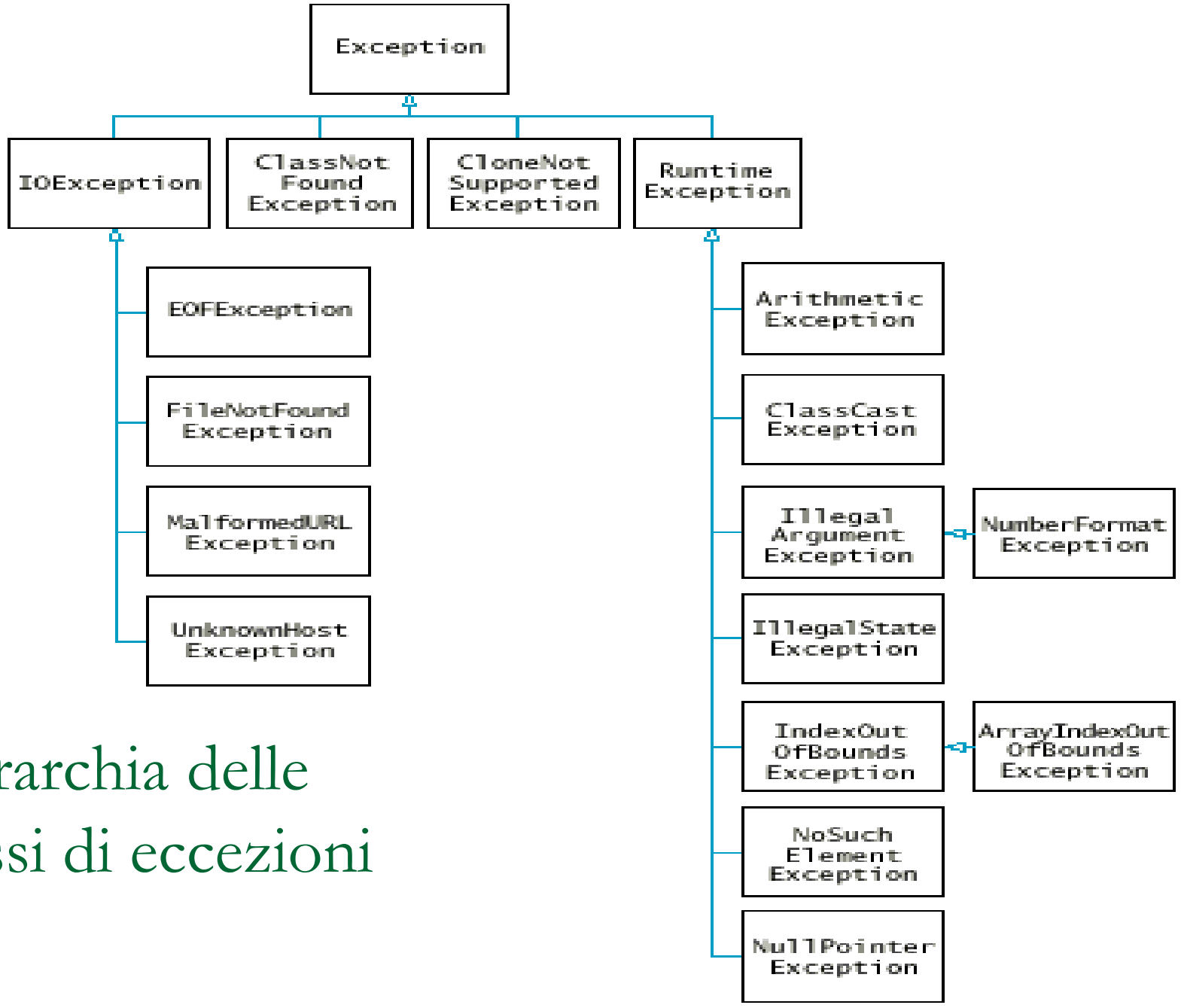
- Casi di errore più comuni, gestibili da programma
- Un'eccezione è un evento che interrompe la normale esecuzione del programma
- Se si verifica un'eccezione il controllo passa ad un gestore delle eccezioni
- Il compito del gestore è:
  - eseguire il codice previsto per il trattamento dell'errore
  - 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



Eccezioni non controllate

- Le eccezioni si distinguono in eccezioni controllate e eccezioni non controllate

- Eccezioni non controllate

  NON è obbligatorio gestire questo tipo di eccezioni

  - segnalano errori evitabili con un'attenta programmazione
  - Es.
    - `NullPointerException`: uso di un riferimento `null`
    - `IndexOutofBoundException`: accesso ad elementi esterni ai limiti di un array

Eccezioni non controllate

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

Eccezioni controllate

- Eccezioni controllate

  E' obbligatorio gestire questo tipo di eccezione

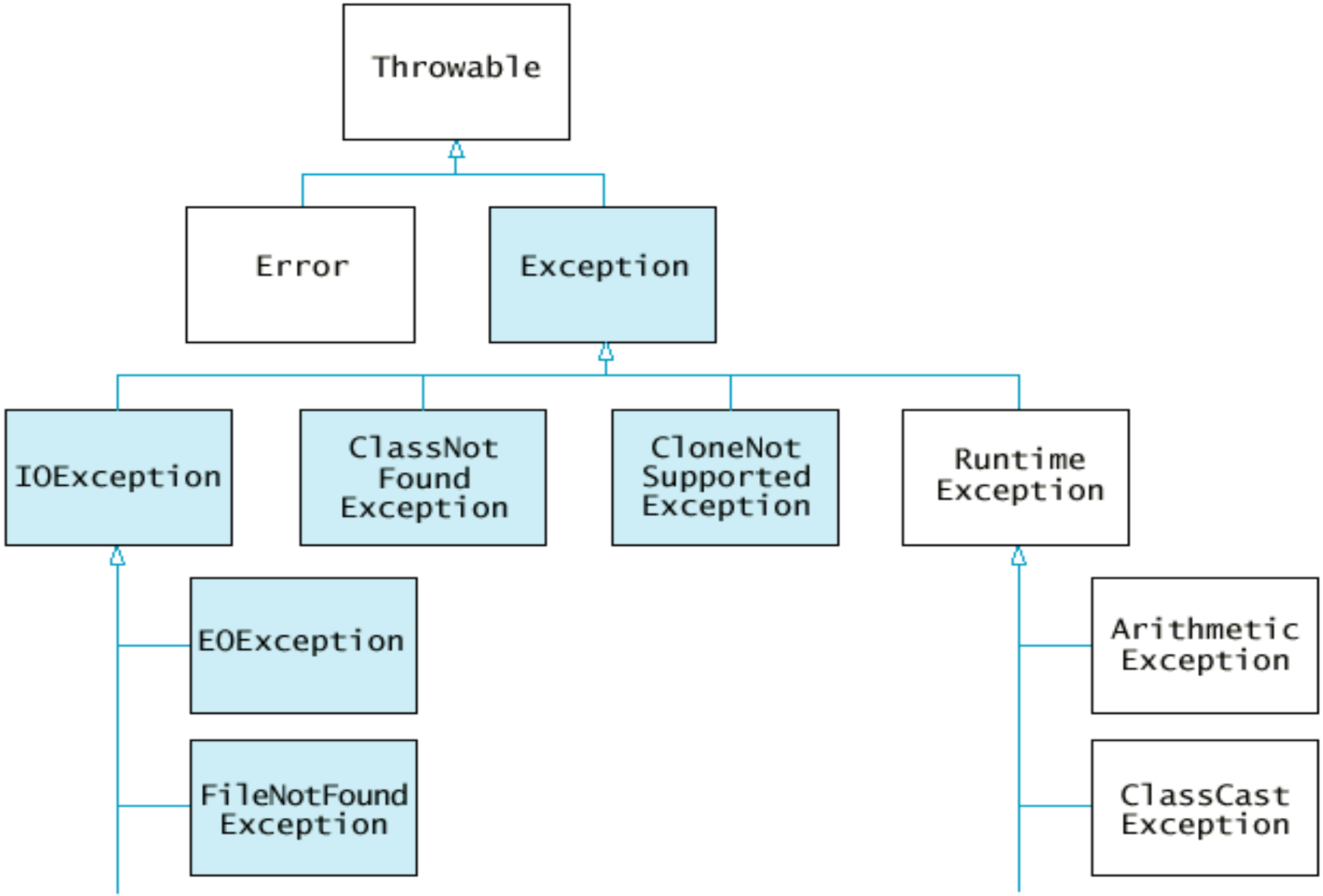
  - errori caustati da eventi esterni (es., errore del disco, interruzione del collegamento di rete,..)
  - situazioni che richiedono l'attenzione del programmatore (es., CloneNotSupportedException)

  - `Es.`
    - `EOFException`: terminazione inaspettata del flusso di dati in ingress
    - `FileNotFoundException`: file non trovato nel file system

Eccezioni controllate

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

Eccezioni controllate e non controllate



Progettare Nuove Eccezioni

- Se nessuna delle eccezioni predefinite è adeguata, possiamo progettarne una nuova.
- Per definire una nuova eccezione si deve estendere un'eccezione esistente
  - eccezione non controllata si estende una classe nella discendenza di `RuntimeException` (in genere viene estesa direttamente questa classe)
  - eccezione 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
- La definiamo come eccezione NON CONTROLLATA

```
public class DivisionePerZeroException extends
                                        RuntimeException{

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

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

Lanciare un'eccezione

- 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

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

Esempio

```
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;
}
```

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();
  }
}
```

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

Gestire le eccezioni

Due possibilità:

- segnalare esplicitamente che il gestore delle eccezioni deve eseguire solo le operazioni di routine

  (clausola throws nella firma dei metodi)

- Inserire un codice alternativo da eseguire

  (utilizzo del costrutto try-catch)

Segnalare eccezioni

- `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 di tipo diverso

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

Catturare le eccezioni

- Le eccezioni se non catturate causano l'arresto del programma

- Per catturare le eccezioni si usa l'enunciato `try`, seguito da tante clausole `catch` quante sono i tipi di eccezione a cui si vuole dare risposta

Catturare eccezioni

```
try
{
    istruzione
    istruzione
        ...
}
catch (TipoEccezione variabile)
{
    istruzione
    istruzione
    ...
}
catch (TipoEccezione variabile)
{
    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 prima clausola `catch` nell'elenco che ha il tipo dell'eccezione lanciata
  - il tipo dell'eccezione lanciata deve essere compatibile con (instance of) il tipo dell'eccezione dichiarato nella clausola

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)
             throws BadDataException
59:       {
```

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: }
```

Esecuzione programma e lancio di eccezioni

- Al lancio dell'eccezione (comando throw new ..) viene interrotta l'esecuzione normale del programma
- Il controllo passa al gestore delle eccezioni (modulo della Java Virtual Machine)
- Se si è all'interno di un modulo try, vengono analizzate le clausole catch
  - Se viene eseguita una clausola catch

    - il gestore delle eccezioni termina

    - il controllo passa al modulo principale della JVM

    (l'esecuzione riprende dalla prima istruzione che segue il costrutto try-catch)

Esecuzione programma e lancio di eccezioni

- Se nessuna delle clausole catch si applica o non si è in un modulo try, viene fatto il return del metodo attualmente in esecuzione (pop dallo stack delle chiamate)
  - Se sono presenti moduli finally vengono eseguiti prima di fare i return
- Se si fa il return dal metodo main senza che l'eccezione venga catturata,

  l'esecuzione del programma viene interrotta

  definitivamente e viene stampato il messaggio di errore.