



UNIVERSITÀ DEGLI STUDI DELL'AQUILA

Laboratorio di Programmazione ad Oggetti

Ph.D. Juri Di Rocco
juri.dirocco@univaq.it
<http://jdirocco.github.io>





Sommario

Eccezioni

- › Introduzione
- › Blocco try/catch
- › Clausola finally
- › Try-with-resources
- › Gerarchia eccezioni
- › Checked e unchecked
- › Keyword throws e throw
- › Overriding dei metodi
- › Creare delle proprie eccezioni
- › Stacktrace



Eccezioni (1)

- › Momento ideale per individuare errori è durante compilazione
- › Non sempre è possibile quindi è necessario un meccanismo durante esecuzione che gestisca errori
- › Linguaggi come C generalmente gestiscono gli errori mediante convenzioni
 - Valore particolare
 - Flag globale che viene esaminato dal destinatario
- › **Definizione**
 - Evento che si verifica durante l'esecuzione di un programma che interrompe il normale flusso delle istruzioni
 - Esempi
 - › Crash dell'HD
 - › Accesso al di fuori dell'array



Eccezioni (2)

- › Esempio: Supponiamo di realizzare una funzione che legge un file e lo pone in memoria
- › Pseudo-codice

```
readFile {  
  open the file;  
  determine its size;  
  allocate that much memory;  
  read the file into memory;  
  close the file;  
}
```



Eccezioni (3)

- › Funzione sembra corretta ma vengono ignorati errori potenziali ovvero
 - Cosa accade se il file non può essere aperto?
 - Cosa accade se la lunghezza del file non può essere determinata?
 - Cosa accade se non vi è abbastanza memoria da allocare?
 - Cosa accade se la lettura fallisce?
 - Cosa accade se il file non può essere chiuso?



Eccezioni (4)

```
errorCodeType 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;
        }
    }
}
```



Eccezioni (5)

.....

```
        close the file;

        if (theFileDidntClose && errorCode == 0) {
            errorCode = -4;
        } else {
            errorCode = errorCode and -4;
        }
    } else {
        errorCode = -5;
    }
    return errorCode;
}
```



Eccezioni (6)

- › Si è passati dalle 7 linee originali a 29 (+400%)
- › Flusso normale del codice si perde all'interno della gestione degli errori rendendo difficile da leggere
- › Maggiore difficoltà nella manutenzione del codice
- › Soluzione
 - Separazione tra codice normale e condizioni di errore



Eccezioni (7)

```
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;  
    }  
}
```



Eccezioni (8)

Passi durante l'**esecuzione** quando si verifica eccezione

- › Creato un oggetto e posto nell'heap
- › Flusso di esecuzione corrente viene interrotto
- › Sistema di run-time ha compito di trovare del codice (tramite lo stack delle chiamate) che gestisce l'eccezione (*exception handler*)
 - Individua corretto exception handler utilizzando tipo eccezione
 - Se non è presente alcun exception handler il sistema di runtime termina il programma



try/catch (1)

- › Si tenta di eseguire il codice e si intercetta un'eccezione si pone rimedio

- › Sintassi

```
try {  
    Istruzioni  
}  
catch (<Tipo eccezione> <Identificatore>) {  
    Altre istruzioni  
}
```

- › L'istruzione `try` identifica un blocco d'istruzioni in cui può verificarsi un'eccezione



try/catch (2)

- › Un blocco `try` è seguito da una o più clausole `catch`, che specificano quali eccezioni vengono gestite
- › Ogni clausola `catch` corrisponde a un **tipo** di eccezione sollevata
- › Quando si verifica un'eccezione, l'esecuzione continua con la prima clausola che corrisponde all'eccezione sollevata



try/catch (3)

› Esempio

```
public class TestEccezione1 {  
    public static void main(String[] args) {  
        int a = 10;  
        int b = 0;  
        int c = a/b;  
        System.out.println(c);  
    }  
}
```



try/catch (4)

```
$ java TestEccezione1  
Exception in thread "main" java.lang.ArithmeticException: / by zero  
    at TestEccezione1.main(TestEccezione1.java:5)
```

- › Tipo Eccezione: `java.lang.ArithmeticException`
- › Messaggio descrittivo: `/ by zero`
- › Metodo dove è stata lanciata l'eccezione:
`TestEccezione1.main`
- › File in cui è stata lanciata l'eccezione: `TestEccezione1.java`
- › Riga in cui è stata lanciata l'eccezione: `5`



try/catch (5)

› Esempio con eccezione

```
public class TestEccezione2 {  
    public static void main(String[] args) {  
        int a = 10;  
        int b = 0;  
        try {  
            int c = a/b;  
            System.out.println(c);  
        } catch (ArithmeticException exc) {  
            System.out.println("Divisione per zero...");  
        }  
    }  
}
```



try/catch (6)

› Esempio con eccezione 2

```
public class TestEccezione3 {  
    public static void main(String[] args) {  
        int a = 10;  
        int b = 0;  
        try {  
            int c = divisione(a, b);  
            System.out.println(c);  
        } catch (ArithmeticException exc) {  
            System.out.println("Divisione per zero...");  
        }  
    }  
    public static int divisione(int a, int b) {  
        return a / b;  
    }  
}
```




try/catch (7)

› Esempio con eccezione 2

```
public class TestEccezione3 {  
    public static void main(String[] args) {  
        int a = 10;  
        int b = 0;  
        try {  
            int c = Math.floorDiv(5, 0);  
            System.out.println(c);  
        } catch (ArithmeticException exc) {  
            System.out.println("Divisione per zero...");  
        }  
    }  
    public static int divisione(int a, int b) {  
        return a / b;  
    }  
}
```



Catch multipli (1)

- › A partire da Java SE 7 è possibile specificare all'interno di una clausola `catch` più tipi di eccezione (**multi-catch**)
 - Si ha un errore in compilazione vi sono due tipi D_i and D_j ($i \neq j$) dove D_i è un sotto-tipo di D_j
- › E' utile quando il contenuto dei blocchi `catch` è identico
- › Riduce duplicazione di codice



Catch multipli (2)

› Esempio

```
int a = 10;
int b = 0;
try {
    int c = a/b;
    System.out.println(c);
} catch (ArithmeticException exc) {
    System.out.println("Divisione per zero...");
} catch (NullPointerException exc) {
    System.out.println("Reference nullo...");
} catch (Exception exc) {
    exc.printStackTrace();
}
```



Catch multipli (3)

```
int a = 10;
int b = 0;
try {
    int c = a/b;
    System.out.println(c);
} catch (ArithmeticException | NullPointerException exc) {
    System.out.println(exc.getMessage());
} catch (Exception exc) {
    exc.printStackTrace();
}
```



Clausola finally (1)

- › Istruzione `try` può avere una clausola `finally` opzionale
- › Se non viene sollevata nessuna eccezione, le istruzioni nella clausola `finally` vengono eseguite **dopo che si è concluso** il blocco `try`
- › Se si verifica un'eccezione, le istruzioni nella clausola `finally` vengono eseguite **dopo le istruzioni della clausola** `catch` appropriata
- › In definitiva se è presente clausola `finally` viene **sempre** eseguita indipendentemente dal verificarsi o meno di un'eccezione
- › Generalmente viene utilizzato per liberare risorse utilizzate all'interno del blocco `try` (es. Files, DB)



Clausola finally (2)

› Esempio

```
public class TestEccezione {  
    public static void main(String[] args) {  
        int a = 10;  
        int b = 0;  
        try {  
            int c = a/b;  
            System.out.println(c);  
        } catch (ArithmeticException exc) {  
            System.out.println("Divisione per zero...");  
        } catch (Exception exc) {  
            exc.printStackTrace();  
        } finally {  
            System.out.println("Tentativo di operazione");  
        }  
    }  
}
```

```
$ java TestEccezione  
Divisione per zero...  
Tentativo di operazione
```



Clausola finally (3)

› Esempio

```
public class TestEccezione {  
    public static void main(String[] args) {  
        int a = 10;  
        int b = 2;  
        try {  
            int c = a/b;  
            System.out.println(c);  
        } catch (ArithmeticException exc) {  
            System.out.println("Divisione per zero...");  
        } catch (Exception exc) {  
            exc.printStackTrace();  
        } finally {  
            System.out.println("Tentativo di operazione");  
        }  
    }  
}
```

```
$ java TestEccezione  
5  
Tentativo di operazione
```



try-with-resources (1)

- › E' una clausola `try` dove vengono dichiarate una o più risorse
- › Una risorsa è un oggetto che deve essere **chiuso** dopo che il programma ha terminato il suo utilizzo
- › Clausola `try-with-resources` garantisce la chiusura alla fine dello statement
- › Ogni oggetto che implementa interfaccia `java.lang.AutoCloseable` può essere usato come risorsa



try-with-resources (2)

› Esempio senza

```
static String readFirstLineFromFileWithFinallyBlock(String path) throws IOException {  
    BufferedReader br = new BufferedReader(new FileReader(path));  
    try {  
        return br.readLine();  
    } finally {  
        if (br != null) br.close();  
    }  
}
```

› Esempio con

```
static String readFirstLineFromFile(String path) throws IOException {  
    try (BufferedReader br = new BufferedReader(new FileReader(path))) {  
        return br.readLine();  
    }  
}
```



try-with-resources (3)

› Esempio senza

- Se i metodi `readLine` e `close` lanciano entrambi eccezioni il metodo `readFirstLineFromFileWithFinallyBlock` lancia l'eccezione avvenuta nel blocco `finally` (`close`). Altra è soppressa

› Esempio con

- Se le eccezioni sono lanciate dal blocco `try` e dallo statement `try-with-resources` (ovvero nel metodo `close()`) il metodo lancia eccezione del blocco `try`
- Eccezione lanciata dal blocco `try-with-resources` viene soppressa
- E' possibile recuperare l'eccezione soppressa con metodo `Throwable.getSuppressed`



try-with-resources (4)

› Esempio risorse multiple

```
public class TryWithResources2 {
    public void selectFromDB() {
        try(Connection conn = DriverManager.getConnection("url",
                                                         "username", "password");

            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery("SELECT * FROM PERSONA ");
            while (rs.next()) {
                System.out.println(rs.getString(1));
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```



try-with-resources (5)

- › Java 9 permette di utilizzare all'interno del blocco try variabili `final` oppure variabili effettivamente non modificate (**effectively final**)

```
public class TestTryWithResource2 {  
    public static void main(String args[]) {  
        RisorsaChiudibile risorsaChiudibile = new RisorsaChiudibile();  
        try (risorsaChiudibile) {  
            System.out.println("Sto per chiudere: " +  
                               risorsaChiudibile);  
        }  
        catch (Exception exc) {  
            exc.printStackTrace();  
        }  
    }  
}
```



Gerarchia (1)

- › In Java un oggetto eccezione è sempre un'istanza di una classe derivata da `Throwable`
- › Gerarchia si suddivide in due categorie
 - `Error`
 - › Errori che si verificano all'interno della VM
 - dynamic linking
 - hard failure
 - › Difficilmente è possibile recuperare da errori di questo tipo
 - › Esempi
 - `OutOfMemoryError`
 - `StackOverflowError`

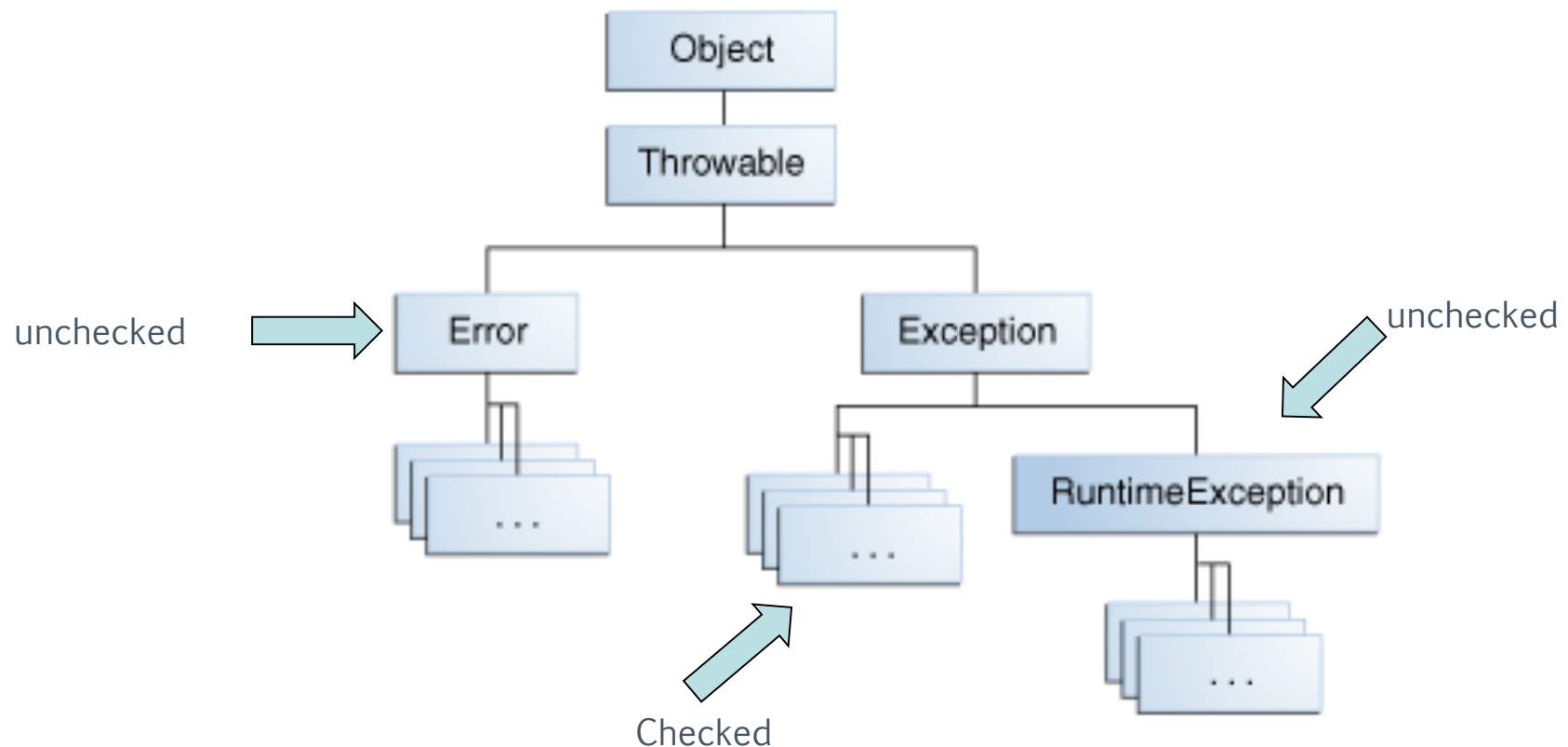


Gerarchia (2)

- Exception
 - › RuntimeException e sue sottoclassi
 - Si verificano quando è stato commesso un errore di programmazione
 - Esempi
 - › Cast definito male: `ClassCastException`
 - › Accesso ad un puntatore nullo: `NullPointerException`
 - › Altre classi che non derivano da `RuntimeException`
 - Si verificano quando si è verificato qualcosa di imprevisto
 - Esempio
 - › Apertura di un file: `FileNotFoundException`
- › Eccezioni `Error` e `RuntimeException` e suoi figli sono dette **unchecked** (non verificate) e non è obbligatorio gestirle
- › Le altre sono dette **checked** (verificate) ed è **obbligatorio** gestirle ovvero è necessario inserirle in un blocco `try/catch` oppure usare clausola `throws` in un metodo



Gerarchia (3)



In realtà anche classi che estendono da `Throwable` e non da `Error` ed `Exception` sono checked



Gerarchia (4)

› Esempio 1

```
public class TestStackOverflow {  
    static int numeroChiamata;  
  
    public static void main(String[] args) {  
        funzioneRicorsiva();  
    }  
  
    public static void funzioneRicorsiva() {  
        System.out.println("Invocazione metodo numero: " +  
                           numeroChiamata++);  
        funzioneRicorsiva();  
    }  
}
```

java.lang.StackOverflowError



Gerarchia (5)

› Esempio 2

```
class Point {  
    int x,y;  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
}
```

```
class TestNullPointerException {  
    public static void main(String[] args) {  
        Point p = null;  
        System.out.println("Accesso variabile d'istanza x di p: " + p.x);  
    }  
}
```



Gerarchia (6)

› Esempio 3

```
class TestCheckedException {  
    public static void main(String[] args) {  
        if (args.length!=1) return;  
        FileReader reader = null;  
        try {  
            reader = new FileReader( args[ 0 ] );  
        } catch(FileNotFoundException e) {  
            System.out.println("File non trovato");  
        } finally {  
            if (reader!=null) {  
                try {  
                    reader.close();  
                } catch (IOException e) { /*Do nothing*/ }  
            }  
        }  
    }  
}
```



Gerarchia (7)

› Esempio 4

```
public class TestMultiCheckedException {  
    public static void main( String[] args ) {  
        if ( args.length != 1 )  
            return;  
        BufferedReader reader = null;  
        try {  
            reader = new BufferedReader( new FileReader( args[ 0 ] ) );  
            String linea = null;  
            while ( ( linea = reader.readLine() ) != null ) {  
                System.out.println( "linea letta = " + linea );  
            }  
        } catch ( FileNotFoundException e ) {  
            System.out.println( "File non trovato!" );  
        }  
    }  
}
```



Gerarchia (8)

```
.....
    catch ( IOException e ) {
        System.out.println( "Eccezione in lettura!" );
    }
    finally {
        if ( reader != null ) {
            try {
                reader.close();
            }
            catch ( IOException e ) {
                //Do nothing
            }
        }
    }
}
```



Gerarchia (9)

› Esempio 5

```
public class TestCheckedException {  
    public static void main(String[] args) {  
        if (args.length!=1) return;  
        try (BufferedReader reader = new BufferedReader( new FileReader( args[ 0 ] ));){  
            String linea = null;  
            while ( ( linea = reader.readLine() ) != null ) {  
                System.out.println( "linea letta = " + linea );  
            }  
        } catch(FileNotFoundException e) {  
            System.out.println("File non trovato");  
        } catch(IOException e) {  
            System.out.println("eccezione IO");  
        }  
    }  
}
```



Clausola throws (1)

- › Eccezioni **checked** possono non essere racchiuse all'interno di un blocco `try/catch`
- › Si utilizza clausola `throws` all'interno della dichiarazione di un metodo
- › Sintassi metodi

`[modificatoriAccesso]`

`[static | abstract | final | native | synchronized]*`

`tipoRitorno nomeMetodo(listaparametri)`

`[throws exceptions]`

- › Nota: `exceptions` è una lista di eccezioni separata da “,”
- › E' il **chiamante** che deve racchiudere invocazione metodo all'interno di un blocco `try/catch`
- › E' possibile catturare l'eccezione e rilanciarla mediante clausola `throw`



Clausola throws (2)

› Esempio 1

```
import java.io.*;

public class TestThrowsKeyword {
    public static void main( String[] args ) {
        if ( args.length != 1 )
            return;
        try {
            readFile( args[ 0 ] );
        } catch( FileNotFoundException e ) {
            System.out.println("File Not Found!");
        } catch ( IOException e ) {
            System.out.println("Errore nel file!");
        }
    }
}
```



Clausola throws (3)

.....

```
private static void readFile( String filename )
    throws FileNotFoundException, IOException {

    BufferedReader reader = new BufferedReader( new FileReader( filename ) );
    String linea = null;
    while ( ( linea = reader.readLine() ) != null ) {
        System.out.println( "linea letta = " + linea );
    }
    if ( reader != null ) {
        try {
            reader.close();
        } catch ( IOException e ) {
            //Do nothing
        }
    }
}
```




Clausola throw (1)

› Esempio 2

```
public class TestThrowKeyword {  
    public static void main( String[] args ) {  
        if ( args.length != 1 )  
            return;  
        try {  
            readFile( args[ 0 ] );  
        } catch( FileNotFoundException e ) {  
            System.out.println("File Not Found nel main!");  
        } catch ( IOException e ) {  
            System.out.println("Errore nel file nel main!");  
        }  
    }  
}
```

.....



Clausola throw (2)

```
.....

private static void readFile( String filename )
    throws FileNotFoundException, IOException {

    BufferedReader reader = null;

    try {

        reader = new BufferedReader( new FileReader( filename ) );

        String linea = null;

        while ( ( linea = reader.readLine() ) != null ) {

            System.out.println( "linea letta = " + linea );

        }

    }

}

.....
```



Clausola throw (3)

```
.....  
  
    catch( FileNotFoundException e ) {  
        System.out.println("File Not Found!");  
        throw e;  
    } catch ( IOException e ) {  
        System.out.println("Errore nel file!");  
        throw e;  
    } finally {  
        if ( reader != null ) {  
            try {  
                reader.close();  
            }  
            catch ( IOException e ) {  
                //Do nothing  
            }  
        }  
    }  
}
```



Overriding metodi (1)

- › Eccezioni checked (verificate) inserite in dichiarazione di metodo hanno impatto nell'overriding
- › Metodo che effettua l'overriding non può lanciare un'eccezione più **generale** di quella del metodo della superclasse
- › Stessa cosa vale per l'implementazione di metodi dichiarati in un'interfaccia



Overriding metodi (2)

› Esempio 1

```
import java.io.IOException;

public class Base {
    public void metodoA() throws IOException {}
}

import java.io.FileNotFoundException;

public class Derivata extends Base {
    public void metodoA() throws FileNotFoundException { }
}

public class AltraDerivata extends Base {
    public void metodoA() throws Exception { } //ERRORE
}
```



Overriding metodi (3)

› Esempio 2

```
import java.io.IOException;

public interface Interfaccia {
    void metodoA() throws IOException;
}
```

```
import java.io.IOException;

public class InterfacciaImpl implements Interfaccia {
    public void metodoA() throws IOException { }
}
```

```
public class AltraInterfacciaImpl implements Interfaccia {
    public void metodoA() throws Exception { } //ERRORE
}
```



Proprie eccezioni (1)

- › E' possibile creare delle proprie eccezioni per indicare condizioni di errore o eccezioni non previste dalla libreria standard di Java
- › E' sufficiente derivare da `Exception` o `RuntimeException` o da qualsiasi altra eccezione



Proprie eccezioni (2)

› Esempio

```
public class MyException extends RuntimeException {  
    public MyException() {  
    }  
    public MyException( String message ) {  
        super( message );  
    }  
    public MyException( String message, Throwable cause ) {  
        super( message, cause );  
    }  
    public MyException( Throwable cause ) {  
        super( cause );  
    }  
}
```




Proprie eccezioni (3)

```
public class TestMyException {  
    public static void main( String[] args ) {  
        if ( args.length != 1 )  
            return;  
  
        try {  
            readFile( args[ 0 ] );  
        } catch ( MyException e ) {  
            System.out.println( "MyException!" );  
        }  
    }  
}
```

.....



Proprie eccezioni (4)

```
.....  
private static void readFile( String filename ) {  
    BufferedReader reader = null;  
    try {  
        reader = new BufferedReader( new FileReader( filename ) );  
        String linea = null;  
        while ( ( linea = reader.readLine() ) != null ) {  
            System.out.println( "linea letta = " + linea );  
        }  
    } catch ( FileNotFoundException e ) {  
        System.out.println( "File Not Found!" );  
        throw new MyException( "MyException", e );  
    }  
}
```

```
.....
```



Proprie eccezioni (5)

```
.....
    catch ( IOException e ) {
        System.out.println( "Errore nel file!" );
        throw new MyException( "MyException", e );
    } finally {
        if ( reader != null ) {
            try {
                reader.close();
            } catch ( IOException e ) {
                //Do nothing
            }
        }
    }
}
```



Stacktrace (1)

- › E' possibile stampare sullo standard di error l'eccezione con il relativo stack delle chiamate

- `public void printStackTrace()`

- › Esempio

```
class MyClass {  
    public static void main(String[] args) {  
        crunch(null);  
    }  
    static void crunch(int[] a) {  
        mash(a);  
    }  
    static void mash(int[] b) {  
        try {  
            System.out.println(b[0]);  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```



Stacktrace (2)

- › E' possibile impostare l'eccezione originale come "causa" della nuova eccezione ovvero si possono annidare
- › Eccezioni standard hanno un costruttore con un parametro di tipo `Throwable` che identifica la causa
 - `public Throwable(String message, Throwable cause)`
- › E' possibile utilizzare metodo `initCause` stesso effetto del costruttore
- › E' possibile recuperare la causa originale mediante metodo `getCause`



Stacktrace (3)

› Esempio

```
class MyClass {  
    public static void main( String[] args ) {  
        crunch( null );  
    }  
    static void crunch( int[] a ) {  
        try {  
            mash( a );  
        } catch ( MyException e ) {  
            e.printStackTrace();  
            System.err.println( "-----" );  
            e.getCause().printStackTrace();  
            System.err.println( "-----" );  
        }  
    }  
}
```

.....



Stacktrace (4)

.....

```
static void mash( int[] b ) {  
    try {  
        System.out.println( b[ 0 ] );  
    }  
    catch ( Exception e ) {  
        throw new MyException( "Errore", e );  
    }  
}  
}
```