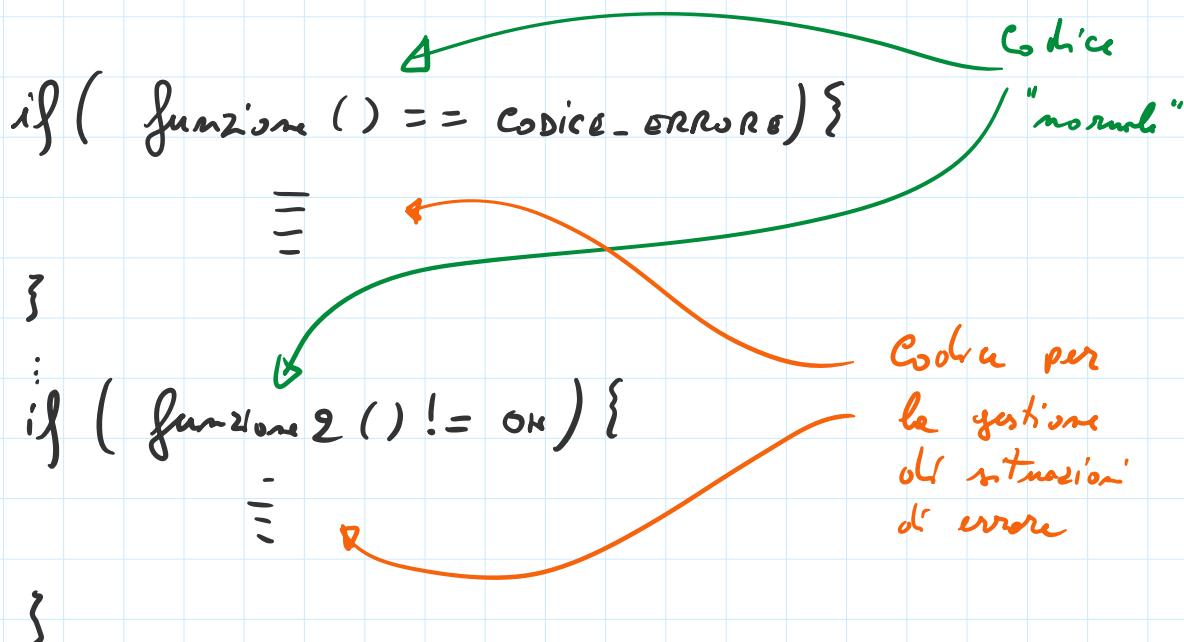


Eccezioni

Mecanismo utile alla rappresentazione di situazioni di errore e alla loro gestione

Senza eccezioni :



Il codice normale è "meccanico" e quello per la gestione degli errori.

Con le eccezioni si può bene mantenere separato il codice "normale" da quello per la gestione degli errori.

Alcune possibili situazioni di errore :

- l'applicazione prova a scrivere in una porzione del file system su cui non ha i diritti di scrittura

- l'applicazione comunica con il mondo esterno

- 2) l'applicazione comunica con il mondo esterno attraverso la rete e c'è una disconnessione
- 3) accedo a un array usando un indice non valido
- 4) chiamo un metodo su un riferimento nullo
- 5) c'è un errore nella jvm (il linking non ha successo)

Nelle situazioni 1) e 2) ha senso prevedere azioni di recupero

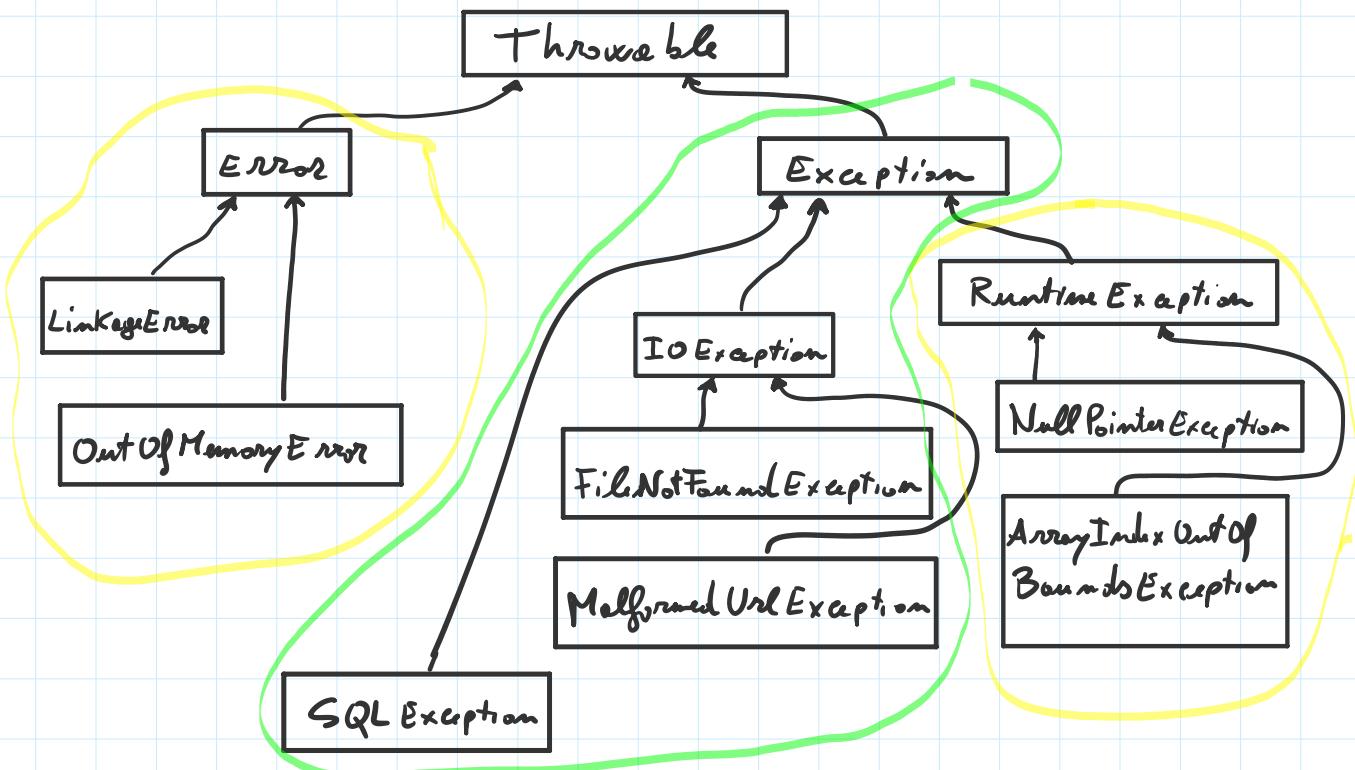
- chiedere all'utente se vuole salvare i dati in un altro punto del file system
- riprovare e connettersi dopo un po' di tempo

Nelle situazioni 3), 4) e 5) non ha senso prevedere azioni di recupero

- 3) e 4) sono errori logici, dobbiamo modificare il codice (i sbaglihi)
- 5) è al di là della sfera di competenza del programmatore

In Java le situazioni di errore sono rappresentate come oggetti eccezione (istanze di opportune classi).

Le eccezioni sono organizzate in una gerarchia:



Eccezioni si dividono in
checked —
unchecked —

Il compilatore obbliga il programmatore a fornire codice di gestione per le eccezioni checked

Non c'è obbligo per le eccezioni unchecked

Due fasi fondamentali nella vita di un'eccezione

- il lancio
- la cattura

lanciamo un'eccezione quando riscontriamo una
situazione anomala (la situazione di errore)

Le eccezioni vengono create da gestori
in cui specifichiamo cosa vogliamo fare (le
azioni di recupero).

Come si lancia un'eccezione?

Con l'istruzione throw

:

```
if (situazione - di - errore) {
```

```
    Exception e = new Exception();
```

```
    throw e;
```

}

:

in genere però le cose si scrivono così

:

```
if (situazione - di - errore)
```

```
    throw new Exception();
```

:

Ho usato Exception per rappresentare

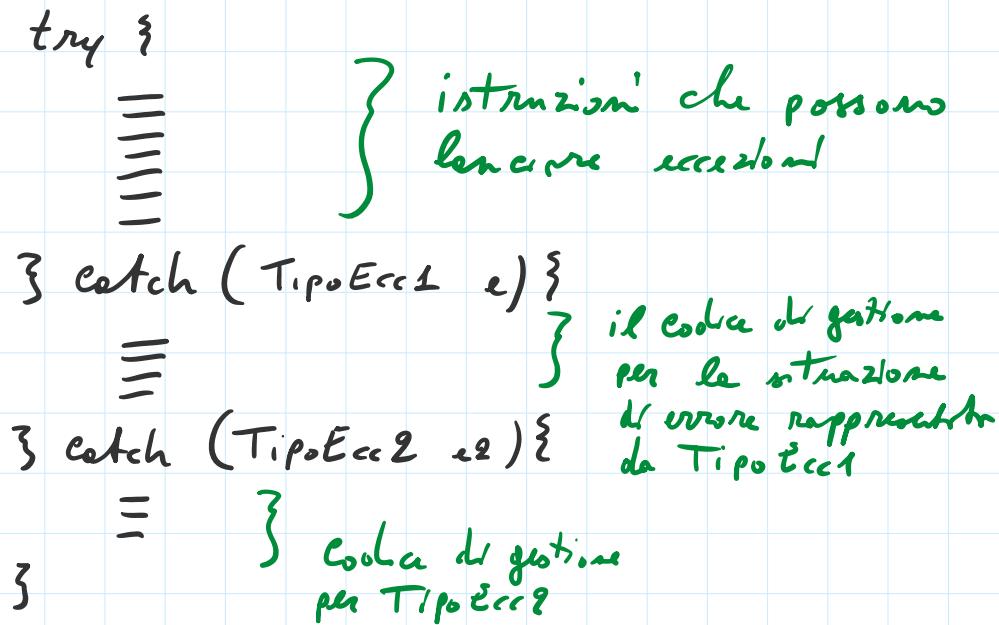
la situazione di errore

- non ho usato Error perché rappresenta error della JVM
- non ho usato Throwable perché troppo generico
- non ho usato RuntimeException perché rappresenta error logici

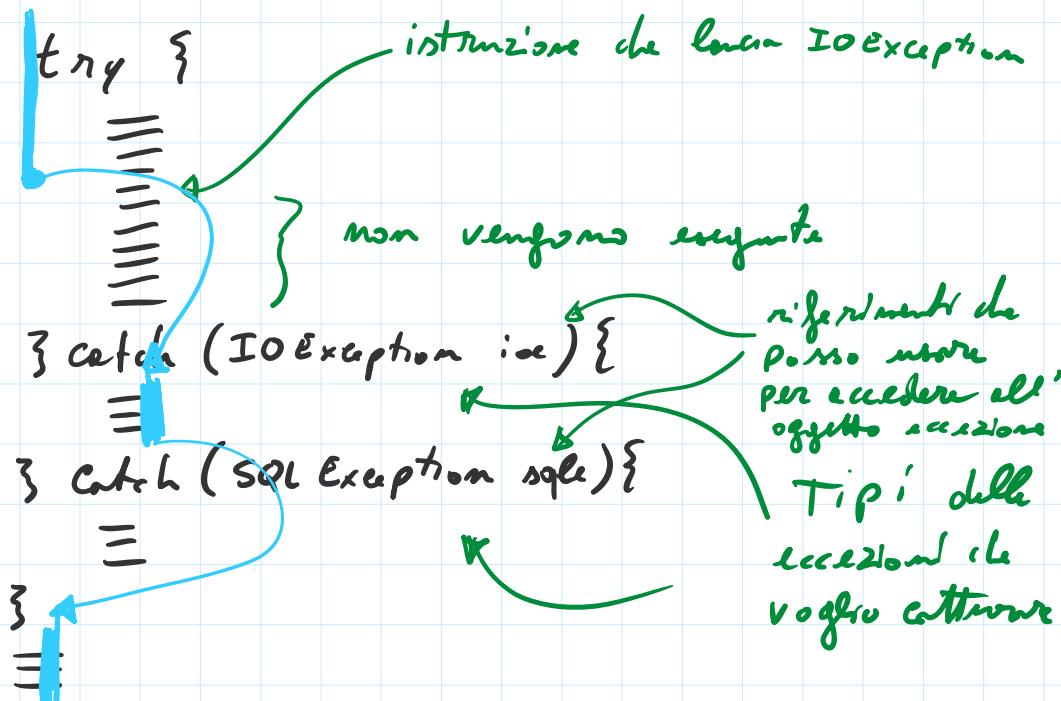
questo riferimento
è non mi serve
e niente quindi



Per catturare e gestire un'eccezione si usa il costrutto try-catch

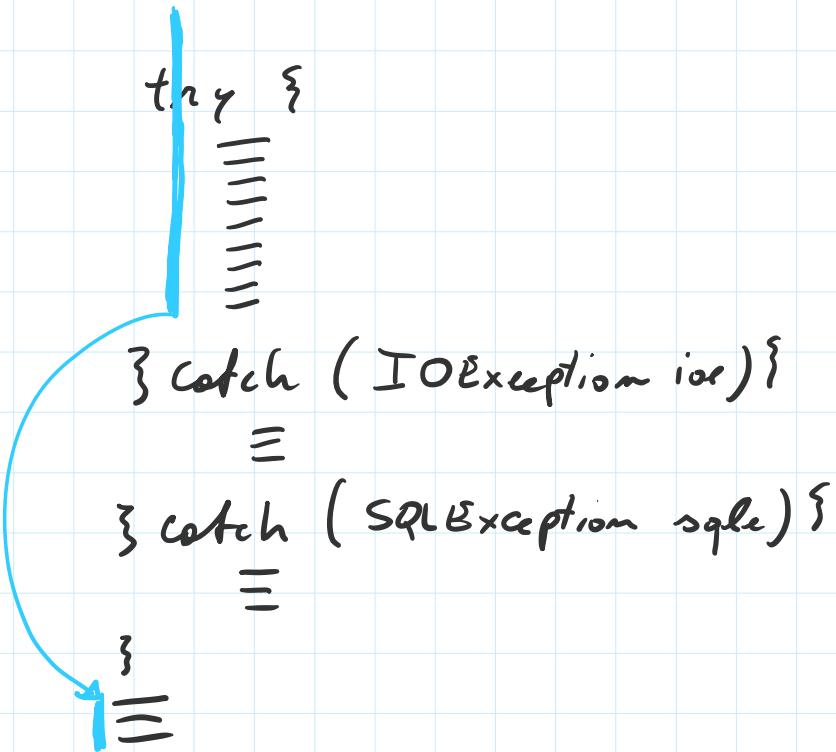


Se un'istruzione nel blocco try lancia un'eccezione l'eccezione salta al blocco catch corrispondente poi vengono eseguite le istruzioni dopo l'ultimo blocco catch



in caso di eccezione "normale"

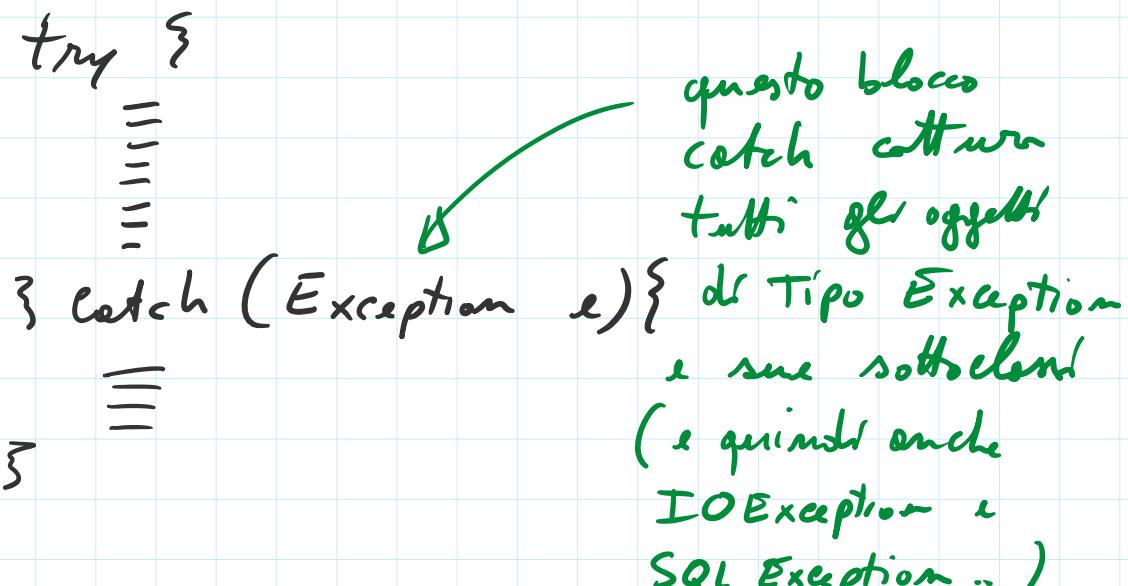
in corso di esecuzione "normale"



Quando viene lanciata un'eccezione viene eseguito il blocco catch corrispondente.

Corrispondente: il tipo dell'oggetto eccezione è un sottotipo di quanto indicato nel blocco catch

Quindi'



IOException e
SQLException ..)

Quando viene lanciata un'eccezione i blocchi catch vengono eseguiti in sequenza

Non appena ne viene trovato uno corrispondente viene eseguito (gli altri, quelli a seguire, non vengono eseguiti).

Questo comporta che non è possibile scrivere i blocchi catch in ordine qualunque quando i tipi di eccezioni catturate sono in una relazione super tipo - sottotipo

- Dobbiamo mettere prima i blocchi catch relativi ai tipi più specifici

try {
= = =

No

} catch (Exception e) {

=

} catch (IOException ioe) {

=

}

non verrebbe
mai eseguito
perché il primo
"cchiappa" anche
IOException

try {
= = =

OK

, dal più

```

    ==
} catch (IOException ioe) {
    ==
} catch (Exception e) {
    ==
}

```

dal più
 specifico
 al meno
 specifico

Nel blocco catch posso accedere all'oggetto eccezione

```

    throw new Exception("Operazione non voluta");
    nel blocco catch:
}

} catch (Exception e) {
  String s = e.getMessage();
  System.out.println(s);
}

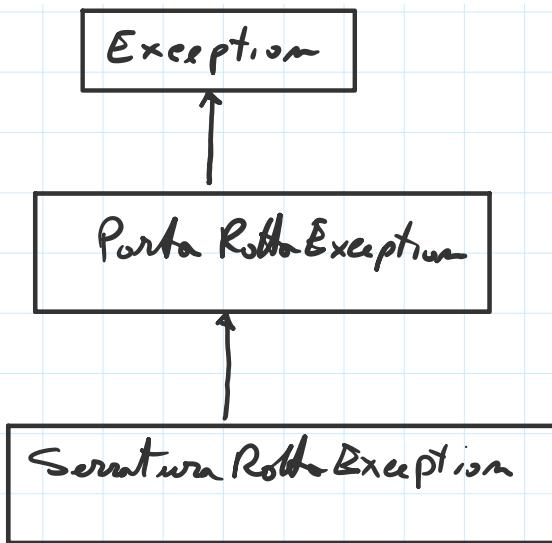
```

Recupero
 il messaggio

Il programmatore può definire nuovi tipi di eccezioni

Tipicamente viene estesa la classe Exception

Esempio:



```

public class PortaRottaException extends Exception {
    public PortaRottaException() {
        super();
    }
    public PortaRottaException(String m) {
        super(m);
    }
}
  
```

```

public class SerraturaRottaException extends
    PortaRottaException {
  
```

```

    public SerraturaRottaException() {
        super();
    }
}
  
```

```

    public SerraturaRottaException(String s) {
        super(s);
    }
}
  
```

Per lanciare :

```
if ( situazione d' errore )  
    throw new PortafoglioException();
```

In effetti il blocco try-catch
può prevedere anche un blocco finally

```
try {  
    ...  
    ...  
}  
} catch (Tipo1 t1) {  
    ...  
}  
} catch (Tipo2 t2) {  
    ...  
}  
} finally {  
    ...  
}  
}
```

} queste istruzioni vengono eseguite sempre
(sia in caso di eccezione
sia se tutto normale)

Se tutto "normale"

```
try {  
    ...  
    ...  
}  
} catch ( — ) {  
    ...  
}  
} catch ( — ) {  
    ...  
}  
} finally {  
    ...  
}
```

> ~~grande~~
≡
≡

in presenza di situazioni anomale

```
try {  
    ...  
} catch (...) {  
    ...  
} catch (...) {  
    ...  
} finally {  
    ...  
}
```

In pratica il codice del blocco finally
viene eseguito sempre
- & spesso contiene del codice di pulizia

```
import java.io.*;  
  
class Main {  
    public static void main(String[] args) {  
        Writer w = null;  
        try {  
            w = new FileWriter("out.txt");  
            for(int i=0; i<10; i++)  
                w.write(String.valueOf(i) + System.getProperty("line.separator"));  
        } catch (IOException ioe) {  
            System.out.println("Errore: " + ioe.getMessage());  
        } finally {  
            try {  
                if(w != null)  
                    w.close();  
            } catch (IOException e){  
                // EAT  
            }  
        }  
    }  
}
```

riparato

```
// EAT  
    }  
}  
}
```

4

questo è uno dei pochi problemi
che non siamo autorizzati a bloccare
perché in cui scrivere un
e catch voto

PA2021 ecc2

The screenshot shows a Java development environment with the following details:

- Files** sidebar:
 - Main.java (selected)
 - jdt.ls-java-project
 - out.txt
- Main.java** code editor:

```
1 import java.io.*;
2
3 class Main {
4     public static void main(String[] args) {
5         Writer w = null;
6         try {
7             w = new FileWriter("out.txt");
8             for(int i=0; i<10; i++)
9                 w.write(String.valueOf(i) + System.getProperty
("line.separator"));
10        } catch (IOException ioe) {
11            System.out.println("Errore: " + ioe.getMessage());
12        } finally {
13            try {
14                if(w != null)
15                    w.close();
16            } catch (IOException e){
17                // EAT
18            }
19        }
20    }
21 }
```
- Console** tab:

```
> javac -classpath .:/run_dir/junit-4.12.jar:/run_dir/hamcrest Q x
1.3.jar:/run_dir/json-simple-1.1.1.jar -d . Main.java
> java -classpath .:/run_dir/junit-4.12.jar:/run_dir/hamcrest-core-1
.3.jar:/run_dir/json-simple-1.1.1.jar Main
> more out.txt
0
1
2
3
4
5
6
7
8
9
> []
```

Attention: non lasciare mai vuoti i blocchi catch!



try {
= ← X
= }

} catch (...) {
}
=

Eccezioni e metodi

Supponiamo che nel corpo di un metodo ci siano istruzioni che possono lanciare eccezioni.

2 possibilità:

- gestiamo la situazione anomala localmente con un blocco try-catch
- propaghiamo l'eccezione al chiamante

Certe eccezioni possono "fuoriuscire" da un metodo (quelle che abbiamo deciso di non gestire localmente).

Le eccezioni che possono fuoriuscire dovranno essere indicate con la clausola throws

public class Pila {
=

 public Object pop() throws PilavuotaException {
 if (empty())
 throw new PilavuotaException();
 }

```
if (empty())
    throw new PilavuotaException();
} = } queste istruzioni non
vengono eseguite se
viene lanciata PilavuotaException
```

La "catena" di chiamate dei metodi
viene percorsa a ritroso fino
a trovare un blocco catch
in grado di catturare l'eccezione

Se un metodo può lanciare più eccezioni
(di tipo diverso) basta specificarle tutte dopo
il throws

```
void m() throws IOException, SQLException {
```

=

§

Il ragionamento vale per le eccezioni
che possiamo ignorare

```
void m1() throws IOException {
```

≡
 try { } se tutte le istruzioni che possono
 generare SQL Exception sono qui
 } catch (SQLException se) { allora non
 possono
fornire errori
 } =
 } =
 } =
 le eccezioni che possono
lanciare IOException
possono essere in
un punto qualunque

Per le eccezioni checked la gestione
è obbligatoria:

- o propagare
- o gestione locale

void salva() throws IOException {
 ≡
 write();
 } = } se lancia eccezione
di tipo IOException
le istruzioni rimanenti
non vengono eseguite.
 } =

Non c'è obbligo per quelle un checked

void v() {
 } =

`int[] x = new int[-];
x[..];
o.m();`

\equiv

3

Esempio

[PA2021_ecc1](#)