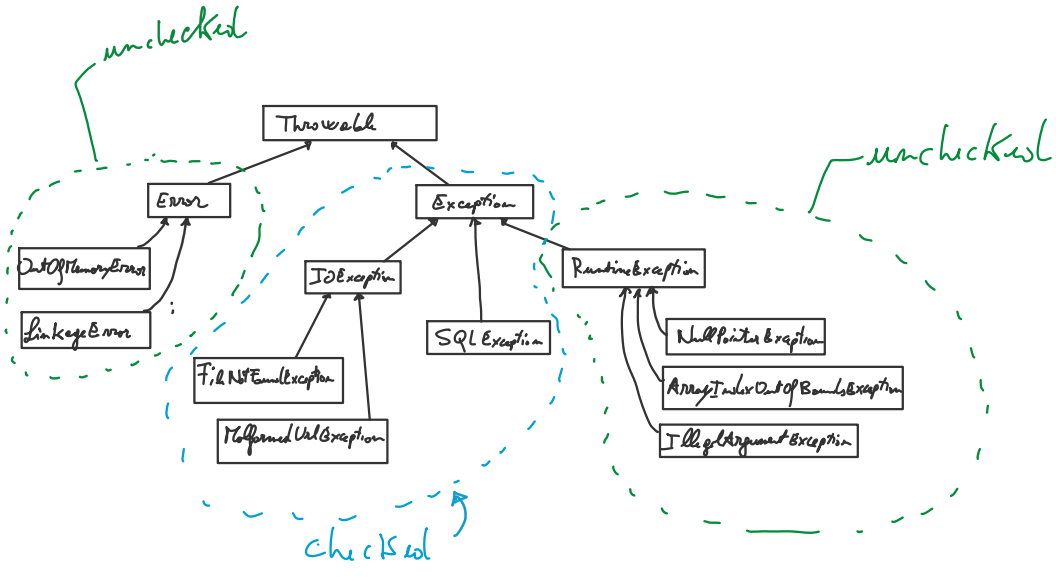
Lezione 8

martedì 30 ottobre 2018 15:39



Due categorie

- eccezioni controllate (checked)
- eccezioni non-controllate (unchecked)

Per le eccezioni checked il programmatore è <u>obbligato</u> a fornire codice di gestione

Per le eccezioni unchecked il programmatore <u>non</u> è obbligato a fornire codice di gestione
- può comunque fornirlo
- ma in genere non lo fa perché
  - si tratta di errori logici (caso 3 e 4 della lista vista in Lezione 7)
  - si tratta di errori della VM (fuori dalla sua portata)

Per lanciare un'eccezione:

```
≡
if (condizione anomala) {
    Exception e = new Exception();
    throw e;
}
≡
```

la forma che comunemente si usa è:

```
≡
if (condizione anomala)
    throw new Exception();
≡
```

} se faccio throw ... queste istruzioni non vengono eseguite, si salta alla prima istruzione del codice di gestione.

Per catturare eccezioni:

```
try {
    ≡
    if (condizione anomala)
        throw new Exception();   // in caso di cond. anomala salto a quest. istruzione si passa a eseguire
    ≡
} catch (Exception e) {
    ≡                            // -> codice di gestione
}
≡                                // al termine del blocco catch si passa a questa istruzione
```

La versione completa del try-catch è:

```
try {
    ≡                            // se viene lanciata un'eccezione si passa a eseguire il blocco catch corrispondente
} catch (TipoEcc1 e1) {
    ≡                            // dopo aver eseguito il blocco catch si passa al blocco finally (se presente) o all'istruzione *
} catch (TipoEcc2 e2) {
    ≡
} catch (TipoEcc3 e3) {
    ≡
} finally {
    ≡                            // viene eseguito sempre (sia per esecuzione normale, sia in caso di errore)
}
*
```

È anche possibile scrivere

```
try {
    ≡
} finally {
    ≡
}
```

Cosa vuole dire "blocco catch corrispondente"?
È corrispondente se il tipo indicato nel blocco catch è dello stesso tipo o è un supertipo dell'eccezione

Questo comporta che posso scrivere

```
try {
    ≡
} catch (IOException e){
    ≡
} catch (Exception i){
    ≡
}
```

non non posso scrivere

```
try {
    ≡
} catch (Exception i) {
    ≡
} catch (IOException e){
    ≡
}
```

i catch vengono esaminati in sequenza

In una situazione come questa il secondo non potrebbe essere mai scelto

È obbligatorio scrivere andando dal più specifico al più generale.

Attenzione non usare mai dei catch "vuoti"

```
try {
    ≡
} catch (Exception e) { }
≡
```

non ci accorgiamo che c'è stato un problema e continuiamo facendo finta di niente (difficile poi trovare la sorgente del problema).

È possibile impostare il messaggio associato a un'eccezione

```
if( condizione)
    throw new Exception("Si è verificato questo errore...");
```

il messaggio può essere recuperato con il metodo getMessage()

```
try {
   ≡

} catch (Exception e){
    System.out.println( e.getMessage());
}
```

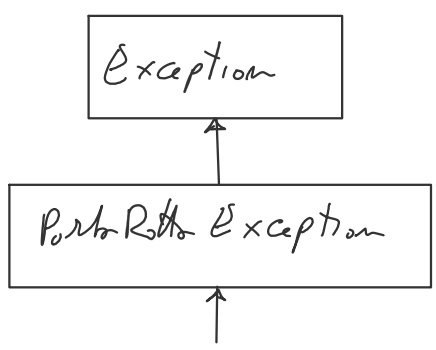
È possibile creare nuovi tipi di eccezioni come sottoclassi di Exception

Esempio:

```
public class PortaRottaException extends Exception {

    public PortaRottaException() {
        super();
    }

    public PortaRottaException(String m){
        super(m);
    }

}
```

è possibile definire dei tipi ancora più specifici

```
public class SerraturaRottaException extends PortaRottaException {

    public SerraturaRottaException () {
        super();
    }

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

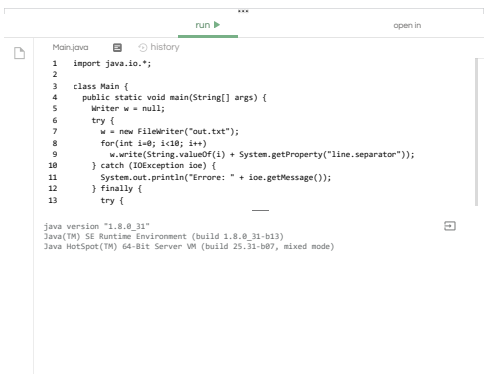


SerraturaRottaException

```
try {
    ≡
    if (portaRotta) throw new PortaRottaException();
    ≡
    if (serraturaRotta) throw new SerraturaRottaException();
    ≡
} catch (SerraturaRottaException e) {
    ≡
} catch (PortaRottaException p) {
    ≡
}
```

Esempio:

PA1819_ecc2




```java
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
      }
    }
  }
}
```

Codice di pulizia, eseguito sempre

Eccezioni e metodi

È possibile lanciare un'eccezione in un metodo e catturarla in un altro (in un punto a monte nella catena di chiamate di metodo aperte e non chiuse)

Un metodo deve indicare quali eccezioni possono "fuoriuscire" quando viene chiamato

- usare clausola throws

```
public class Pila {
        ≡
        public Object pop() throws PilaVuotaException {
                ≡ ↖ qui ci deve essere del codice
        }           che può generare quel tipo di
        ≡           eccezione
                        if (vuota())
}                           throw new PilaVuotaException()
```

Se possono fuoriuscire più tipi di eccezioni:

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

Cosa vuol dire "fuoriuscire"?

```
void m1() throws IOException {
        ≡
        try {
            ≡                    } istruzioni che possono lanciare SQLException
        } catch (SQLException s){}
            ≡
        }
        ≡
}
```

Qui può essere lanciata IOException; non gestita localmente, propagata

Se in un punto del corpo di un metodo viene lanciata un'eccezione, il resto del corpo non viene eseguito. Si torna al chiamante e si ragiona nello stesso modo.

Dal punto di vista del chiamante, quando invoco un metodo che può lanciare eccezioni:

- racchiudo la chiamata in un blocco try-catch (gestisco l'eccezione localmente)
- la propago al chiamante (devo inserire la clausola throws nel metodo che sto scrivendo)
- la "mappo" in un altro tipo (faccio il catch di quell'eccezione e mi ci di altro tipo, con corrispondente clausola throws)

```
void f1() {
    ≡
    try {
      ≡
      f2();
      ≡
    } catch (IOException e) {
        ≡
    }
}
```

l'eccezione è gestita localmente

```
void f2() throws IOException {
    ≡
    ≡
    ≡
}
```

Codice che può lanciare IOException

```
void f1() throws IOException {
    ≡
    f2();
    ≡
}
```

propago

```
void f2() throws IO
    ≡
}
```

Co
pu
IO

```
void f1() throws MiaBcc {
    ≡
    try {
      ≡
      f2();
      ≡
    } catch (IOException i) {
        throw new MiaBcc(i.getMessage());
    }
    ≡
```

f2 come sop.

>

↑ lo dobbiamo fare per le eccezioni controllate (checked)

Possiamo farlo anche per quelle non controllate

- ma in genere non lo si fa

```
void   g()  {
        ≡
        v[index] = ---
        ≡
}
```

Esempio

PA1819_ecc1

```
1   class Main {
2
3     public static void metodo1() throws Exception {
4       System.out.println("Entro in metodo 1");
5       try {
6           System.out.println("try di metodo1");
7           if(Math.random() < 1.0)
8             throw new Exception("ABC");
9           System.out.println("try di metodo1 bis");
10      } finally {
11        System.out.println("Dentro finally di metodo1");
12      }
13      System.out.println("Fine di metodo1");
```

```
java version "1.8.0_31"
Java(TM) SE Runtime Environment (build 1.8.0_31-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.31-b07, mixed mode)
```

Asserzioni

- Utili in fase di sviluppo
- Diverse da eccezioni:

  eccezioni: situazioni anomale che possono ragionevolmente verificarsi a tempo di esecuzione e da cui vogliamo recuperare

  asserzioni: utili solo in fase di sviluppo
  controllare condizioni che sappiamo essere vere

Due forme

```
assert expression1;

assert expression1 : expression2;
```

expression1 è la condizione che deve essere vera
(se non lo è, c'è un problema nel codice)

Se expression1 è falsa viene generato un AssertionError

expression2 contiene dei dettagli utili a capire cosa è successo
(il valore di expression2 viene usato per riempire l'AssertionError)

```
assert x == y : "x = " + x + ", y = " + y;
```

expression1 — expression2

```
assert x == y;
```

Normalmente la JVM viaggia con le asserzioni disabilitate

Possono essere abilitate con le opzioni

- -ea
- -enableassertions

Posso usarle per verificare che l'esecuzione non raggiunga determinati punti del codice

```
void m() {
    =
    if (—)
        return
    =
    if (--)
        return
    ≡
    if (—)
        return
    assert false; ←
}
```

Utile per verificare delle "invarianti":

```
public Contenitore {
        ≡
    public void inserisci (Valore v) {
        int s = getSize();
        <Codice inserimento vero e proprio>
        int t = getSize();
        assert s+1 == t;
    }
        ≡
}
```

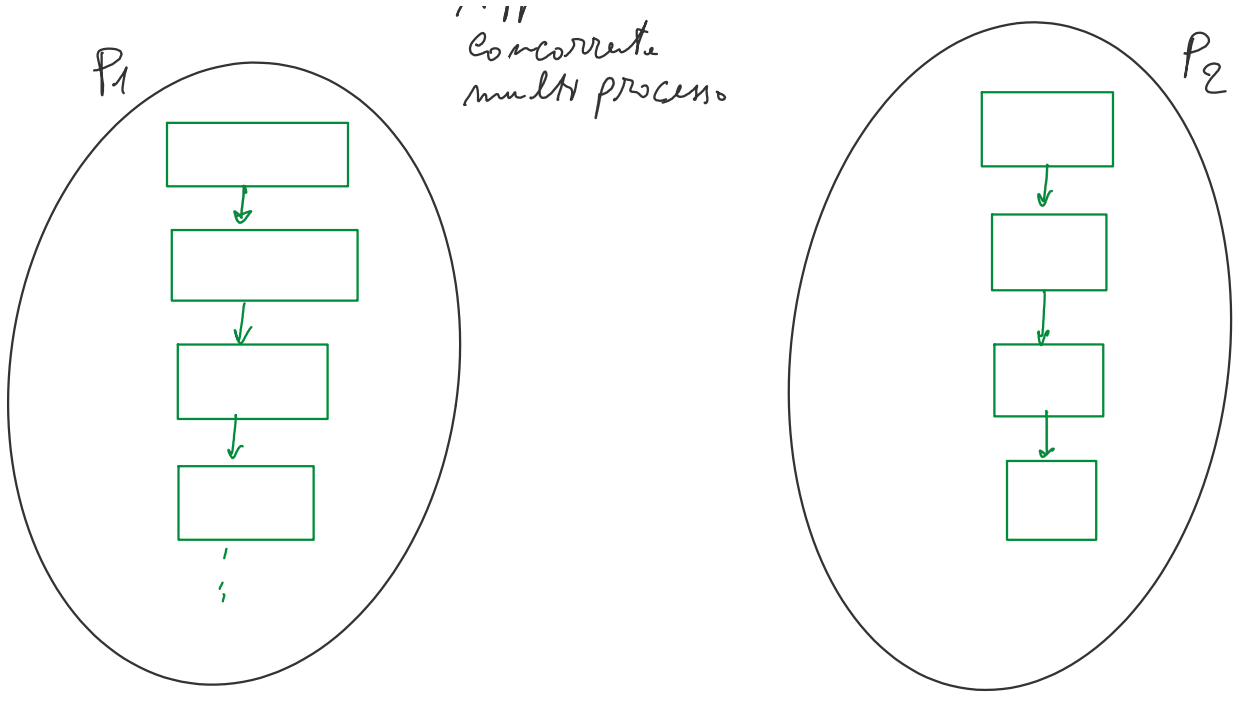
Thread

Processo: flusso di esecuzione + spazio di indirizzamento

Thread: flusso di esecuzione

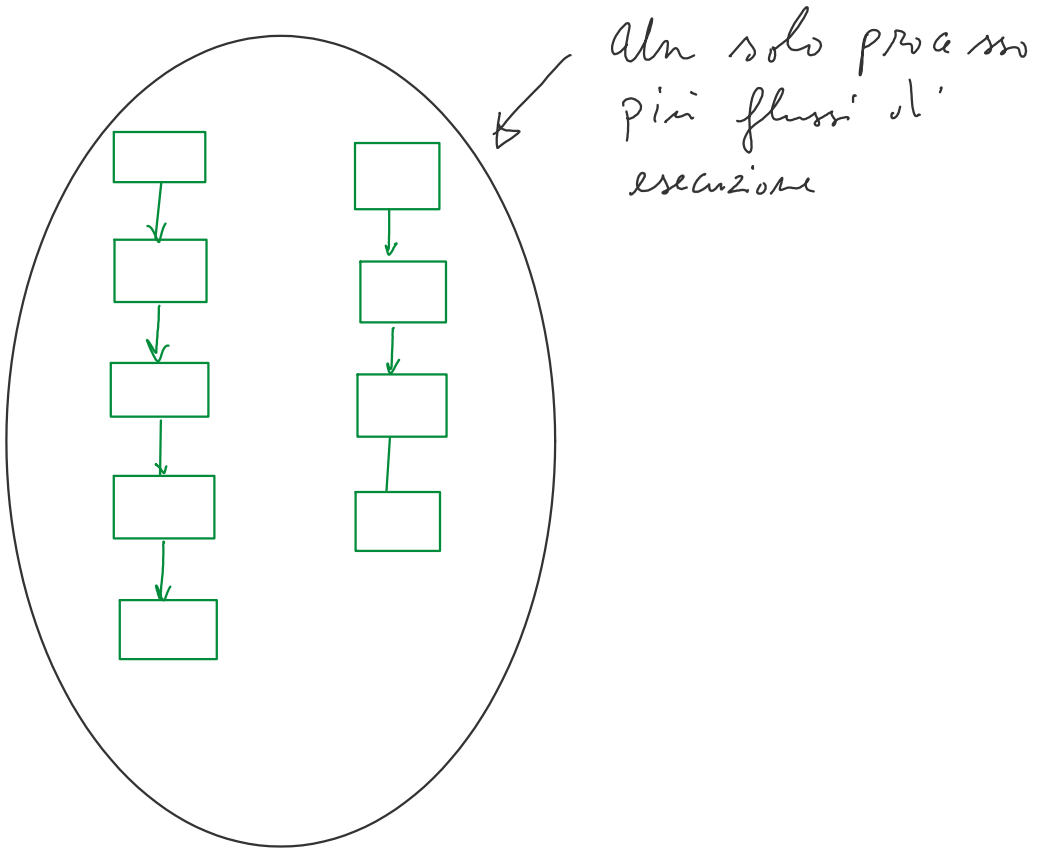
Più thread possono vivere all'interno dello stesso processo

Luca M



Concorrente multi processo

Applicazione concorrente multi-thread



- All'avvio della JVM c'è un solo thread che esegue il corpo del main della classe indicata da riga di comando. (main thread)
- Il main thread può generare dei thread aggiuntivi:
  - in effetti ogni thread ne può generare di nuovi.
- In Java i thread sono rappresentati da oggetti' istanze della classe Java.lang.Thread
- Dobbiamo scrivere delle sottoclassi di Thread ridefinire il metodo run()

creare istanze della classe che estende Thread
avviare i flussi di esecuzione con start()

PA1819_thread1