

Sommario: 26 Aprile, 2016

- Eccezioni: Definizione, Sollevamento, Mascheramento
- ADT e Modificatori
- Interfacce: Segnature, Gerarchia di Classi e Interfacce
- Polimorfismo e Generalizzazione dei tipi coinvolti.

Definition

Una eccezione è un evento, che occorre durante l'esecuzione, e interrompe il normale flusso (i.e. sequenza di controllo) del programma.

- **Meccanismo delle Eccezioni** Si compone di 3 parti
 - **Definizione** delle eccezioni gestite dal programma
 - **Sollevamento** e generazione di un'eccezione
 - **Gestione** e possibile risoluzione dell'eccezione

- **Definizione** delle eccezioni gestite dal programma
 - Eccezioni = Oggetti che Java associa a eventi che interrompono il flusso de programma

```
if (factorial(x) > y) {...
```

quando factorial non calcola un intero positivo, il flusso è interrotto

- **Sottoclassi** di `Java.lang.Exception`

- `Java.lang.Exception` contiene 5 costruttori, con argomenti di tipo diverso, per raccogliere informazioni sullo stato
- Ma possiamo aggiungerne di nuovi (vedi esercizi)
- useremo solo `Exception()` e `Exception(String)`

```
class IllegalArgsException extends Exception{  
    public IllegalArgsException(String s){super(s);}  
}  
class EmptyValueException extends Exception{  
}
```

- `IllegalArgException` permette di specificare il metodo invocato;
- `EmptyValueException` permette di esprimere eccezioni senza altre informazioni
- La scelta dipende dal trattamento che faremo di tale eccezione

- **Sollevamento** di un'eccezione di una classe E di eccezioni
 - Per generare un'eccezione di una classe E si usa il costrutto:
throw new E(...)

- Genera un'eccezione della classe E
- Termina l'invocazione corrente del metodo con tale "sollevamento"

```
public void swap(myIntMutSeq q) throws IllegalArgumentException{
    /* element swapping di this.val con q.val */
    if (q==null) throw new IllegalArgumentException("swap");
    {
        int temp = val();
        valUpdate(q.val());
        q.valUpdate(temp);
    }
}
```

- Le classi E1, ..., En di eccezioni sollevabili devono essere dichiarate nell'intestazione di ogni classe e di ogni metodo che le possono sollevare, utilizzando:
throws E1, ..., En

- **Gestione** di un'eccezione è trattata dal metodo invocante in 3 forme possibili:
 - **Ri-sollevamento** quando non è in grado di (i.e. non sa come) trattarla
 - **Mascheramento** quando è in grado di trattarla e risolverla completamente
 - **Mista** quando è in grado di trattare solo alcuni aspetti. È una combinazione delle due.
- Esaminiamo i 3 casi nella seguente struttura generale:

- Esaminiamo i 3 casi nella seguente struttura generale:

```
class E extends Exception {
    public E(String s){super(s);
}
class E1 extends E {
    public E1(String s){super(s);
}
class E2 extends E {
    public E2(String s){super(s);
}
class A {
    ...
    void callee(C x) throws E{
        ...
        ... throw new E1("Callee: caso semplice...");
        ...
        ... throw new E2("Callee: caso complesso...");
        ...
    }
    //...
}
class B {
    ...
    void caller(){
        ...//altri statements
        ... o.callee(v) ...//statement in cui occorre l'invocazione
        ...//altri statements
    }
}
```

dove abbiamo:

- una gerarchia di eccezioni: $E > E1, E2$
- un metodo callee che genera e solleva eccezioni
- un metodo caller che invoca callee

- **Ri-sollevarmento.** Consiste nel:

- Dichiarare **anche** il metodo caller sollevante eccezioni di E
- Lo statement ‘...o.callee(v)...’ rimane **invariato**
- Quando o.callee(v) solleva un'eccezione E:
caller termina e solleva, **a sua volta**, tale eccezione

```
class A {  
    ...  
    void callee(C x) throws E {  
        ...  
        throw new E1("M1: caso semplice");  
        ...  
        throw new E2("M1: caso complesso");  
        ...  
    }  
    //...  
}  
class B {  
    ...  
    void caller() throws E {  
        ...//altri statements  
        ... o.callee(v) ...//statement in cui occorre l'invocazione  
        ...//altri statements  
    }  
}
```

- **Mascheramento.** Richiede uno **specifico costrutto try-catch**:

```
try{regione di codice mascherata}
catch{eccezione intercettata}{trattamento dell'eccezione}
...
catch{eccezione intercettata}{trattamento dell'eccezione}
```

formato da 2 sezioni:

- Regione di Codice Mascherata:
 - Racchiude il codice "critico"
- Lista Casi di Eccezione Trattati:
 - Discrimina i casi possibili, e
 - Fornisce un trattamento risolutivo

- **Mascheramento. try-catch:**

- Racchiude il codice "critico"
- Discrimina i casi possibili
- Fornisce un trattamento risolutivo

```
class A {
    ...
    void callee(C x) throws E{
        ...
        throw new E1("callee: caso semplice...");
        ...
        throw new E2("callee: caso complesso...");
        ...
    }
    //...
}
class B {
    ...
    void caller() throws E {
        ...//altri statements
        try {
            ... o.callee(v) ...
        }
        catch (E1 e1) {
            System.out.println("generata E1 in "+e1.getMessage()+" risolta in caller");
            //...codice per il trattamento
        }
        catch (E2 e2) {
            System.out.println("generata E2 in "+e2.getMessage()+" risolta in caller");
            //...codice per il trattamento
        }
        ...//altri statements
    }
}
```

Meccanismo delle Eccezioni: Mista

- Mista. Combinazione di Risolleamento e Mascheramento
- Un caso concreto.

```
public class eccezioni1{
//OVERVIEW: collezione di metodi stand_alone

public static Object access(Object a_vector, Object a_integer)
    throws NoVectorException, NoIntegerException, OutOfVectorBounds {
    Vector V;
    try {V = (Vector) a_vector;}
    catch (Exception e) { throw new NoVectorException("access:arg1"); }
    int i;
    try {i = ((Integer) a_integer).intValue();}
    catch(Exception e) { throw new NoIntegerException("access:arg2"); }
    if (V.size() > i) return V.elementAt(i);
    throw new OutOfVectorBounds();
}

// Compile e provate l'esecuzione con il seguente main.

public static void main(String[] args) throws Exception {
    Vector My = new Vector();
    My.addElement("la Prova ha successo");
    My.addElement("Una seconda stringa");
    My.addElement(new Integer(5));
    System.out.println(access(My,My));
}
}
```

Definition

Una interfaccia è la segnatura di una collezione di metodi correlati

- Ha struttura sintattica simile a quella di classe ma contiene solo: Intestazioni di metodi di istanza.

```
public interface DLinkSeq {  
    public DLinkSeq pred();  
    public DLinkSeq succ();  
    public int val();  
    public void valUpd(int v);  
}
```

- Fondamentale Struttura nella Gerarchia delle classi e nella nozione di sottotipo.
- Una classe può essere sottoclasse di una o più interfacce
- Una classe è sottoclasse di un interfaccia se ...

Gerarchia di classi e interfacce

- Una classe può essere sottoclasse di una o più interfacce
- Una classe è sottoclasse di un'interfaccia se ...
 - Dichiarare di "implementare" l'interfaccia
 - Fornisce un **overriding** per ogni metodo dell'interfaccia
 - Un caso concreto.

```
interface sortable{
    public boolean ord(sortable s);
}

class sortedName implements sortable{
    private String name;
    public sortedName(String n){
        name = n;
    }
    public boolean ord(sortable s){
        return ((sortedName)s).name.startsWith(this.name);
    }
}
```

Example

ord in sortedName è riflessivo, e transitivo. Estenderlo in un preordine totale ovvero in modo tale che per ogni u, v , $u.ord(v)$ oppure $v.ord(u)$.

La portata di un identificatore (classe, metodo, costruttore, variabile, campo) in Java non è sufficiente per la sua accessibilità.

- Modificatori di Accesso: stabiliscono l'accessibilità di un identificatore all'interno della sua portata
- 4 tipi di accesso:
 - default (omesso): package.
 - **public**: mondo.
 - **private**: classe.
 - **protected**: package e sottoclassi.
- L'uso dei soli `private` e `public` permette di definire classi che si comportano come ADT.

ADT: Implementazione

- **private**: Ogni campo della classe (stato dei valori) e ogni metodo non nella segnatura dell'ADT
- **public**: I soli metodi della segnatura dell'ADT
- Un esempio concreto è lo ADT delle doubly-linked sequence, sotto

```
public class intMutSeq {
    /* ----- Stato oggetti intMutSeq ----- */
    private intMutSeq Pred;
    private intMutSeq Succ;
    private int Val;

    /* ----- Operazioni: Costruttore ----- */
    /* intMutSeq() è alias di empty() ----- */
    /* ----- cambieremo quando sapremo usare le eccezioni ----- */

    public intMutSeq(intMutSeq p, int v){//alias di add
        /* inserisce dopo p */
        Pred = p;
        Val = v;
        if (p!=null){...}
    }

    /* ----- Metodi per OPERAZIONI ----- */
    /* ----- OSSERVATORI PER ACCESSO COMPONENTI ----- */

    public intMutSeq pred(){...};
    public intMutSeq succ(){...};
    public int val(){...};
    /* ----- MODIFICATORI COMPONENTI ----- */

    public void valUpdate(int v){...};
}
```

- La segnatura di un ADT può essere definita con un'interfaccia
- Implementata da una classe che implementa l'interfaccia

```
interface DLinkSeq{
    public DLinkSeq pred();
    public DLinkSeq succ();
    public int val();
    public void valUpd(int v);
}

public class intMutSeq implements DLinkSeq {
    /* ----- rappresentazione interna ----- */
    private intMutSeq Pred;
    private intMutSeq Succ;
    private int Val;
    public intMutSeq(intMutSeq p, int v){
        Pred = p;
        Val = v;
        if (p!=null){
            Succ = p.Succ;
            p.Succ = this;
            if (Succ!=null) Succ.Pred = this;
        }
    }
    /* ----- Implementazione Metodi ----- */

    public intMutSeq pred(){
        return Pred;
    };
    public intMutSeq succ(){
        return Succ;
    };
    public int val(){
        return Val;
    };
    public void valUpd(int v){
        Val = v;
    };
};
```

- Java contiene varie forme di polimorfismo (Object, Generic, Subtype)
- Qui consideriamo il polimorfismo generico presente anche in altri linguaggi con lo scopo di:
- Generalizzare i tipi ai quali sono applicabili le definizioni introdotte nel programma.

```
class coppia {A left; B right;}
```

- Permette di riusare una stessa definizione invece di doverne definire una per ogni specifico tipo.

```
class coppia {A left; B right;}  
class coppia {B left; A right;}  
class coppia {C left; D right;}
```


- Il Polimorfismo Generico richiede di:
 - Estendere il sistema dei tipi, includendovi **variabili di tipo**
 - Permettere definizioni che sono parametriche su variabili di tipo

```
class coppia<T1,T2> {T1 left; T2 right;}
```

- Permette di riusare una classe parametrica, coppia<T1,T2>, come classe:

```
coppia<A,B> x = new coppia<A,B>()  
coppia<B,A> y = ...  
coppia<C,D> y = ...
```

- per introdurre oggetti, o variabili di tipo diverso e inconfrontabile

- Vediamo un primo esempio di coppia generica:

```
class pair <A,B> {
    A left;
    B right;
}
class Main{
    public static void main(String[] argv){
        pair<Integer,Integer> x = new pair<Integer,Integer>();
        x.left = new Integer(5);
        x.right = new Integer(7);
    }
}
```

- Una diversa definizione di coppia generica per usare conversioni int-Integer fornite dal compilatore

```
class pair <A,B> {
    A left;
    B right;
    pair(A x, B y){
        left = x;
        right = y;
    }
}
class Main{
    public static void main(String[] argv){
        pair<Integer,Integer> x = new pair<Integer,Integer>(5,7);
    }
}
```