

Java: Basilari di Programmazione in Grande

Sommario: 9 Maggio, 2018

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

Interfaccia

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

```
class A extends B implements C1,...,Cn{...}
```

- e i suoi oggetti hanno le interfacce come super-tipo:

```
public void met(C1 x)...
```

- 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).

3/23

interfacce: Metodologie di Uso

- **Vincolano il comportamento** degli oggetti definiti

```
class A implements DLinkSeq{...}
```

- La classe A deve fornire definizioni per i metodi `pred()`, `succ()`,...

- **Garantiscono comportamenti** degli oggetti usati

```
public void met(C1 x)...
```

- il valore legato ad `x` ha tutti i metodi dell'interfaccia `C1`

- **Forniscono API** nella realizzazione di Tipi Astratti

```
class A implements DLinkSeq{...}
```

- La classe A (opportunamente definita) è un ADT per `DLinkSeq`

Gestione delle Eccezioni

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

Meccanismo delle Eccezioni: Definizione

- **Definizione** delle eccezioni gestite dal programma
 - Eccezioni = Oggetti che Java associa a eventi che interrompono il normale flusso di controllo del programma

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

quando factorial non calcola un intero positivo, non abbiamo un valore da restituire all'invocante e quello che sarebbe stato il normale flusso di controllo deve cambiare

- **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 IllegalArgumentException extends Exception{  
    public IllegalArgumentException(String s){super(s);}  
}  
class EmptyValueException extends Exception{  
}
```

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

Meccanismo delle Eccezioni: Sollevamento

- **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);
    }
}
```

Sollevamento: Eccezioni Checked ed Eccezioni Unchecked

- **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(...)
- 2 tipi di eccezioni sollevabili
 - **Checked**
 - Le classi E1, ..., En di eccezioni sollevabili devono essere dichiarate nell'intestazione di ogni metodo che le può sollevare, utilizzando:
throws E1, ..., En
 - Le eccezioni sollevabili fanno parte della signature
 - Il compilatore ne controlla la consistenza nelle invocazioni
 - **Unchecked** – (Noi Non Useremo)
 - Da dichiarare come sottoclasse di RuntimeException
class E extends RuntimeException{...}
 - Il compilatore non controlla la consistenza delle segnature

Meccanismo delle Eccezioni: Gestione

- **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:

Meccanismo delle Eccezioni: Gestione - Caso 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 E1, E2{
        ...
        ... 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

Meccanismo delle Eccezioni: Gestione - Risolleivamento

- **Ri-solleivamento.** 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 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 E1, E2{
        ...
        ... throw new E1("callee: caso semplice...");
        ...
        ... throw new E2("callee: caso complesso...");
        ...
    }
    //...
}
class B {
    ...
    void caller() throws E1, E2 {
        ...//altri statements
        ... o.callee(v) ...//statement in cui occorre l'invocazione
        ...//altri statements
    }
}
```

Meccanismo delle Eccezioni: Gestione - Mascheramento

- **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

Meccanismo delle Eccezioni: Gestione - Mascheramento/2

- **Mascheramento. try-catch:**

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

```
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 E1, E2{
        ...
        ... throw new E1("callee: caso semplice...");
        ...
        ... throw new E2("callee: caso complesso...");
        ...
    }
    //...
}
class B {
    ...
    void caller(){
        ...//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 Risollevamento e Mascheramento

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

Modificatori di Accesso e ADT

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**: universo.
 - **private**: classe.
 - **protected**: package e sottoclassi. – (**Noi Non Useremo**)
- 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){...};
}
```


ADT: Segnatura

- La segnatura di un ADT può essere definita con un'interfaccia
- Implementata da una classe che implementata 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;
    };
}
```

Polimorfismo Generico

- 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.
- Permettere il riuso una stessa definizione invece di definirne una per ogni specifico tipo.

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

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

Polimorfismo Generico: Variabile di Tipo e Parametro

- 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> z = ...
```

- Introdurre oggetti, o variabili di tipo diverso e inconfrontabile

```
coppia<A,B> y = ...  
coppia<A,C> z = ...  
//ma a z non posso assegnare y anche quando C sia superclasse di B
```

Polimorfismo Generico: Esempi

- 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);
    }
}
```

Applichiamo: Valori, Classi, Tipi Astratti, Polimorfismo

Vediamo una storia: Un inizio e la sua evoluzione (vedi fig. Stack)

- **Inizio.** Vogliamo valori IMMUTABLE di tipo stack bounded (a non più di 100 interi):
Definiamo la classe StackImm: implementata con oggetti Array
- **Non Piace.** Implementazione Space Expensive :
Definiamo la classe StackImm2: implementata con oggetti Vector < Integer >
- **Fragile.** Codice esposto a crash e a intrusioni e violazioni:
Definiamo la classe StackImm2ADT: implementata con API e ADT
- **Uso Limitato.** Opera solo con gli int e gli Integer:
Definiamo la classe StackImm2ADTP: implementata con API e ADT polimorfi

Stack: Una brutta definizione di un tipo di dato

```
class Error extends RuntimeException{
}
public class StackImm{
    public int[] p;
    public int n;
    public int top;
    //metodi
    public StackImm(){//synonym of create for introducing empty stacks
        p = new int[100];
        n = 0;
        top = -1;
    }
    public StackImm push(int k){
        if (n==100) throw new Error();
        StackImm ret = new StackImm();
        for (int i=0;i<n;i++)ret.p[i]=p[i];
        ret.p[n]=k;
        ret.n = n+1; ret.top = top+1;
        return ret;
    }
    public int top(){
        if(n==0) throw new Error();
        return p[top];
    }
    public StackImm pop(){
        if (n==0) throw new Error();
        StackImm ret = new StackImm();
        for (int i=0;i<n-1;i++)ret.p[i] = p[i];
        ret.n = n-1; ret.top = top-1;

        return this;
    }
    public boolean empty(){
        return (n==0);
    }
}
```

Applichiamo: Valori, Classi, Tipi Astratti, Polimorfismo /2

... e la storia continua

- **Non Adeguata.** Uguaglianza Valori, Visita Componenti...

Uguaglianza di valori IMMUTABLE:

$x \neq y$ anche se $AF(x) \equiv AF(y)$

Visita:

- È 15 contenuto nello stack1?
- Posso vedere (e stampare nell'ordine) i componenti di uno stack?

- **MUTABLE.** Vogliamo valori MUTABLE di tipo stack bounded (a non più di K interi):

Definiamo la classe StackMutADT: implementata con API e ADT