Completamenti e Tipi Astratti:

Sommario: 15 Maggio, 2019

- **Completamenti:**
  > Polimorfismo di Sottotipo (Lezione15)
- **Il Tipo Astratto Immutable `Stack` in Java:**
  > Struttura del Package per ...
  > Modificatori Private per ...
  > Stato: Struttura MultiComponente
  > Le eccezioni
  > Le operazioni: `waitUp`, `addAll`, `remove`
- **Immutable vs. Mutable in Java (e OCaml).**
- **Il Tipo Astratto Mutable `Stack` in Java:**
  > Struttura del Package, Modificatori Private
  > Stato: Struttura MultiComponente, oppure ...
  > Le eccezioni
  > Le operazioni: `waitUp`, `addAll`, `remove`
- **Esercizi**

Polimorfismo di Sottotipo

- **Polimorfismo Generico**.
  > Variabili di Tipo quantificate Universalmente sui Tipi: `Pair<A,B>`
  > Riuso del codice `Pair` su ogni istanza della coppia di tipi `A` e `B`

- **Polimorfismo Sottotipo**.
  > Variabili di Tipo Vincolate: `OrdPair<A extends ExA, B>`
  > Riuso del codice `Pair` **solo** su ogni istanza di `A` sottotipo di `ExA`

```
class ordPair<A extends Comparable<A>, B extends Comparable<B>> extends Pair<A,B>
                    implements Comparable<ordPair<A,B>>{
    public int compareTo(ordPair<A,B> o){
        if (L.compareTo(o.L) == 1) return 1;
        if (L.compareTo(o.L) == 0) return R.compareTo(o.R);
        return -1;
    }
}
```

- **Vantaggi**. Il codice `OrdPair` può utilizzare le operazioni dei SottoTipi,
  > e diventa parametrico rispetto ai metodi di `A` e `B`, SottoTipi di `ExA` ed `ExB`

- **In Questo Caso**: `Comparable<T>` è un interfaccia per tipi di valori ordinabili.
  > Introduce un metodo per il confronto, `compareTo`, su ciascun membro;
  > Il codice `OrdPair` ora può usare i due metodi `compareTo`;
  > Il codice `OrdPair` è come un codice High-Order:
    + dove i metodi `compareTo` non sono parametri trasmessi
    + ma trovati come metodi dei sottotipi di `A` e `B` utilizzati

Polimorfismo di Sottotipo: Esempio OrdPair<A,B>

- **Polimorfismo Sottotipo**.

```
package ordPair;
import java.lang.*;
/*
public interface Comparable<T>{//interfaccia di di java.lang
    public int compareTo(T o);
        //compareTo(o) > 0 se this maggiore di o
        //compareTo(o) == 0 se this uguale o
        //compareTo(o) < 0 se this minore o }
*/

class Error extends RuntimeException{}
class Pair<A,B>{A L; B R;}
class ordPair<A extends Comparable<A>, B extends Comparable<B>> extends Pair<A,B>
                    implements Comparable<ordPair<A,B>>{
    public int compareTo(ordPair<A,B> o){
        if (L.compareTo(o.L) == 1) return 1;
        if (L.compareTo(o.L) == 0) return R.compareTo(o.R);
        return -1;
    }
}
```

- **CompareTo**: è un interfaccia per tipi di valori ordinabili.
  - > Introduce un metodo per il confronto `compareTo` su ciascun membro;
  - > Il codice `OrdPair` ora può usare i due metodi `compareTo`;
  - > Il codice `OrdPair` è come un codice High-Order:
    - + dove i metodi `compareTo` non sono parametri trasmessi
    - + ma trovati come metodi dei sottotipi di `A` e `B` utilizzati

OrdPair<A,B>: Esempio di Uso

- **Polimorfismo Sottotipo**.

```
class Error extends RuntimeException{}
class Pair<A,B>{A L; B R;}
class ordPair<A extends Comparable<A>, B extends Comparable<B>> extends Pair<A,B>
                    implements Comparable<ordPair<A,B>>{
    public int compareTo(ordPair<A,B> o){
        if (L.compareTo(o.L) == 1) return 1;
        if (L.compareTo(o.L) == 0) return R.compareTo(o.R);
        return -1;
    }
}

class Main{
    public static void main(String[] args) throws Error{
        ordPair<String,Integer> p1 = new ordPair<String,Integer>();
        p1.L = "A"; p1.R = 0;
        ordPair<String,Integer> p2 = new ordPair<String,Integer>();
        p2.L = "a"; p2.R = 0;
        System.out.println("A".compareTo("a"));
        System.out.println(p1.compareTo(p2));
        System.out.println("is p1 GT p2? " + (p1.compareTo(p2)==1));
        }
}
```

Polimorfismo di Sottotipo: Esercizio con `OrdSet<A>`

- **Polimorfismo Sottotipo**.
  > Variabili di Tipo Vincolate: `OrdSet<A extends Exp>`
  > Riuso del codice `Set` **solo** su ogni istanza di `A` sottotipo di `Exp`

```
public class OrdSet<A extends Comparable<A>> extends Set<A>{
    private A max;
    private A min;
    }
    public OrdSet<A> union(Set<A> y){...}
    public A min(){...}
    public A max(){...}
}
```

- **Vantaggi**. Il codice `Set` può utilizzare le operazioni del SottoTipo `Exp`:
  > Operazione possibilmente overridden dalle classi della gerarchia;
  > Il codice `Set` diventa parametrico rispetto alle operazioni del SottoTipo;

- **Applicazioni**: Definizione di Codice High-Order.
  > I metodi di `Set` possono usare le operazioni del SottoTipo come parametri impliciti (High-Order);

Polimorfismo di Sottotipo: Esercizio con SortSeq<A>

- **Polimorfismo Sottotipo**.
  - > Variabili di Tipo Vincolate: `SortSeq<A extends Exp>`
  - > Riuso del codice `Seq` **solo** su ogni istanza di `A` sottotipo di `Exp`

```
package SortSeq;
import java.lang.*;
/*
 Completare:
 Classe Seq<A> con un tipo di dato per sequenze polimorfe di valori
 di tipo A ad accesso diretto con costruttore:
        public Seq(Vector<A> seq);
 e con operazioni per:
        public void add(A x)
        public void scambia(int i, int j);
        public String toString();
*/

class Error extends RuntimeException{}
class Seq<A>{...}
class SortSeq<A extends Comparable<A>> extends Seq<A> {
    public void qSort(){
        ...
    }
}
```

Immutable Stack: Testo per un ADT

```
Definire un ADT per oggetti che forniscano valori Non-Modificabili da
utilizzare in modo simile ai valori stack bounded di int introdotti
da StackImm in slide di Lezione15.
Questo ADT deve trattare stack polimorfi e fornire le seguenti opera-
zioni (con segnatura funzionale)
   push: A -> Stack<A> -> Stack<A>
   toString: Stack<A> -> String
   top: Stack<A> -> A
   pop: Stack<A> -> Stack<A>
   size: Stack<A> -> int
   isin: A -> Stack<A> -> boolean
   waitUp: A ->Stack<A> -> int          (quanti prima)
   addAll: Vector<A> -> Stack<A> -> Stack<A>
   remove: A -> Stack<A> -> Stack<A>    (rimuove prima occorrenza)
In aggiunta ad un costruttore che dato un intero n>0, costruisce uno
Stack<A> con bound n.

Prologo: Intero sviluppo in package di nome StackPack
Prologo: Creare 4 file:
   1 per la API STACK<A>
   1 per la classe Stack<A>
   1 per le eccezioni, newExc
   1 per i codici di tests.
(a) Fornire API
(b) Fornire Stato Concreto, AF e C
(c-d-e-f) Sviluppo:
   (c) Fornire successiva operazione nell'ordine e
       ad ogni definizione procedere con (d),(e),(f)
   (d) Completare la segnatura con le eccezioni sollevate
   (e) Aggiungere definizione classi di eccezioni relative
   (f) compilare ed eseguire test di uso
```

Immutable `Stack`: Struttura del Package per...

- Uso di package per: Delimitare Scope delle Classi
  - non (dichiarate) `public`: Modificatori di Accesso

- Per convenienza di progettazione:
  - 1 directory (folder) in cui raccogliere codice sorgente e codice oggetto
    - C.Sorgente in 4 file:
      - API.java contiene l'API del Tipo Astratto
      - Stack.java contiene un ADT del Tipo Astratto
      - newExc.java contiene le classi di Eccezione
      - Tests.java contiene le classi dei test di uso
    - C. Oggetto: files .class nel package StackPack

  - 1 package, k file X k classi dichiarate `public`
    - 2 file per 2 classi public
    - 2 file per n classi con modificatore default

Interfaccia: API<A>

- Interfaccia:
  - Molti metodi: Alcuni Fondamentali
  - Segnatura: Tipo calcolato API<A>, Eccezioni Utilizzate

```
package StackPack;
import java.util.*;

public interface API<A>{
    public API<A> push(A x) throws FullStackException;
    public String toString();
    public A top() throws EmptyStackException;
    public API<A> pop() throws EmptyStackException;
    public int size();
    public boolean isin(A x);
    public int waitUp(A x)throws NoSuchElementException;
    public API<A> addAll(Vector<A> vv)throws FullStackException;
    public API<A> remove(A x);
}
```

Modificatori Private e Stato MultiComponente

- Modificatori Private per:
  - fields e metodi dell'ADT non dichiarati public nell'API
  - In Stack, i soli fields dello stato

```
public class Stack<A> implements API<A>{
    private A elm;
    private Stack<A> next;
    private int size;
    private int max;
```

- Stato MultiComponente:
  - `elm` ha tipo generico `A`;
  - Ricorsiva sul secondo componente `next`;
  - `size` non è necessario ma è conveniente;

Stato Concreto: AF, I e Costruttori

- AF, I e un solo costruttore che riceve il bound

```
private A elm;
private Stack<A> next;
private int size;
private int max;
/*
 AF(c) = [] iff (c.size = 0)
 AF(c) = [v1,...,vn] iff (c.size > 0)&&(c.size = n)&&(vn = elm)&&
                          (AF(c.next) = [v1,...,vn-1])
 I(c) = (c.size >= 0)&&(c.size <= c.max)&&
        (c.size > 0 => c.elm != null) &&
        (c.size > 1 => c.next != null) &&
        (c.next != null => (c.next.max = c.max &&
                            c.next.size = c.size-1))
 */
public Stack(int n){
    if (n<0) max = 0;
    else max = n;
    size = 0;
}
```

Eccezioni: Le abbiamo considerate tutte?

- Abbiamo introdotto e definito alcune sottoclassi di Exception
- Tra cui `FullStackException`, sotto.

```
public Stack<A> push(A x)throws FullStackException{
    if (size == max) throw new FullStackException("push",max);
    Stack<A> ret = new Stack<A>(max);
    ret.size = this.size+1;
    ret.elm = x;
    ret.next = this;
    return ret;
}
```

- Ma cosa succede quando invochiamo?:
  `o.push(null);` — Aggiungiamo `null` come valore di `elm`!
- E quando lo selezioniamo come `elm`?:
  `elm.toString();` — Solleviamo eccezione `NullPointerException`

- `NullPointerException` (e `ClassCastException`) è sottoclasse di `RunTimeException`
  > Non deve essere dichiarata `throws` nella segnatura;
  > È risollevata automaticamente.
  > Ma conviene, quando possibile, non farla sollevare:
  `if (x == null) return this;`

Le operazioni: `waitUp` e `addAll`

- `waitUp` Lascia che sia `isin` a trattare il caso `x == null`
- `waitUp` usa l'additional `equals` per l'uguaglianza
- `waitUp` è ricorsiva sulla struttura del secondo componente

```
public int waitUp(A x)throws NoSuchElementException{
    if (!isin(x)) throw new NoSuchElementException("waitUp");
    if(elm.equals(x)) return 0;
    return (1 + next.waitUp(x));
}
public Stack<A> addAll(Vector<A> vv)throws FullStackException{
    if (vv == null) return this;
    if (vv.size()+size > max) throw new FullStackException("addAll",size);
    Stack<A> temp = this;
    for(int i=0; i<vv.size(); i++){
        temp = temp.push(vv.get(i));
    }
    return temp;
}
```

- `addAll` controlla il caso `vv == null` e lo risolve.
- `addAll` controlla se lo stack può essere esteso con ... in caso contrario solleva
- `addAll` itera `temp.push(vv.get(i))` per estendere lo stack.
- Structure Sharing: `addAll` crea una struttura che condivide la struttura `this`.

Le operazioni: `remove`

- `remove` restituisce `this` quando `x == null`
- `remove` usa l'additional `equals`: Cosa cambia se `x.equals(elm)`?
- `remove` è ricorsiva sulla struttura del secondo componente

```
public Stack<A> remove(A x){
    if ((x == null)||(size == 0)) return this;
    if (elm.equals(x)) return next;
    Stack<A> ret = new Stack<A>(max);
    ret.elm = elm;
    ret.next = next.remove(x);
    ret.size = ret.next.size() + 1;
    return ret;
}
```

- `remove` applica `new Stack<A>()` prima di attraversare `next`, creando una copia dell'elemento al quale assegna come `next` il valore `this.next.remove(x)`

- `remove` scopre l'effettiva size (i.e. c'era un tale elemento) dopo il calcolo di `this.next.remove(x)` e lo assegna `ret.next.size()+1`

Immutable vs. Mutable in Java.

- **Valori**: Immutable e Mutable, hanno operazioni simili (nello scopo) ma comportamento molto diverso tra loro.

- **Valori Mutable**: possono essere modellati solo in presenza di:
  - Linguaggio di Programmazione con Stato e
  - Assegnamento per Locazioni e Fields di oggetti
  - Allocazione dinamica di Memoria

- **Valori Immutable**: Esprimibili sia in OCaml sia in Java
- **Valori Mutable**: SI in Java, NO in OCaml (funzionale)

- **Valori Mutable**: Confrontiamo gli Stack Immutable in Java con gli Stack Mutable sempre in Java con:
  - Stesso Stato di Rappresentazione
  - Operazioni corrispondenti

Mutable Stack<A> in Java: Package, Modificatori e Stato

- **Package**: Organizzato identicamente alla versione Immutable.
- **Modificatori**: Uso di Private e Public identico nei due casi
- **Stato**: Possiamo Usare ed Usiamo stesso Stato Concreto (anche se non sempre sia la scelta migliore)

```
package StackPack;

import java.io.*;
import java.util.*;

public class Stack<A> implements API<A>{
    private A elm;
    private Stack<A> next;
    private int size;
    private int max;
    /*
    AF(c) = [] iff (c.size = 0)
    AF(c) = [v1,...,vn] iff (c.size > 0)&&(c.size = n)&&
                            (vn = elm)&&
                            (AF(c.next) = [v1,...,vn-1])
    I(c) = (c.size >= 0)&&(c.size <= c.max)&&
           (c.size > 0 => c.elm != null) &&
           (c.size > 1 => c.next != null) &&
           (c.next != null => (c.next.max = c.max &&
                               c.next.size = c.size-1))
    */
```

Mutable Stack<A> in Java: Interfaccia API

- **API**: I PRODUTTORI diventano MODIFICATORI.
  - `push` trattato come un Modificatore
- **API**: Costruttori e Osservatori mantegono la signature
- **API**: Anche le eccezioni sollevabili rimangono

```
package StackPack;
import java.util.*;

public interface API<A>{
    public void push(A x) throws FullStackException;
    public String toString();
    public A top() throws EmptyStackException;
    public void pop() throws EmptyStackException;
    public int size();
    public boolean isin(A x);
    public int waitUp(A x)throws NoSuchElementException;
    public void addAll(Vector<A> vv)throws FullStackException;
    public void remove(A x);
}
```

Mutable Stack<A> in Java: Le operazioni /1

```
public class Stack<A> implements API<A>{
    private A elm;
    private Stack<A> next;
    private int size;
    private int max;
    /* AF(c) = ...
       I(c) = ... */
    public Stack(int n){
        if (n<0) max = 0;
        else max = n;
        size = 0;
    }
    public void push(A x)throws FullStackException{
        if (x == null) return;
        if (size == max) throw new FullStackException("push",max);
        Stack<A> ret = new Stack<A>(max);
        ret.elm = elm;
        ret.next = next;
        ret.size = size;
        elm = x;
        next = ret;
        size++;
    }
    public String toString(){
        if (size == 0) return "[ ]";
        if (size == 1) return "[" + elm.toString() + "]";
        String ret = next.toString();
        ret = ret.substring(0,ret.length()-1);
        return (ret + ", " + elm.toString() + "]");
    }
    public A top()throws EmptyStackException{
        if (size == 0) throw new EmptyStackException("top");
        return elm;
    }
```

Mutable Stack<A> in Java: Le operazioni /2

```
public class Stack<A> implements API<A>{
    private A elm;
    private Stack<A> next;
    private int size;
    private int max;
    ...
    public int size(){
        return size;
    }
    public boolean isin(A x){
        if (size == 0) return false;
        return (elm.equals(x) || next.isin(x));
    }
    public int waitUp(A x)throws NoSuchElementException{
        if (!isin(x)) throw new NoSuchElementException("waitUp");
        if(elm.equals(x)) return 0;
        return (1 + next.waitUp(x));
    }
    public void addAll(Vector<A> vv)throws FullStackException{
        if (vv == null) return;
        if (vv.size()+size > max) throw new FullStackException("addAll",size);
        for(int i=0; i<vv.size(); i++){
            push(vv.get(i));
        }
    }
    public void remove(A x){
        if (x == null || size == 0) return;
        if (elm.equals(x)){
            try {
                pop();
                return;
            }catch(EmptyStackException e){/*impossibile*/}
        }
        next.remove(x);;
        size = next.size() + 1;
    }
```

Esercizi

- Perchè nella programmazione Funzionale `List.fold_left` è considerato un iteratore? Si spieghi cosa è valutato ad ogni iterazione e come variano gli eventuali indici.
- Si mostri come possiamo utilizzare `List.fold_left` per calcolare in OCaml il minimo in una lista di interi.
- Nel presentare il polimorfismo di sottotipo abbiamo introdotto la classe `OrdSet`. Questa classe estende la classe `Set<A>` per generica `A` vincolata ad essere sotto-tipo di `comparable<A>`. La nuova classe estende lo stato concreto con 2 fields interi. Si chiede di definire AF ed I della nuova classe.
- Nel presentare il polimorfismo di sottotipo abbiamo introdotto la classe `OrdSet`. Questa classe estende la classe `Set<A>` per generica `A` vincolata ad essere sotto-tipo di `comparable<A>`. La nuova classe apparentemente ha il solo costruttore di default che come tale non richiede di essere dichiarato.
  (a) Si discuta sotto quali condizioni ciò è accettabile alla luce del fatto che la nuova classe deve fornire i metodi `min` e `max` per il calcolo del minimo e massimo valore contenuto nell'insieme.
  (b) Si aggiunga un costruttore per l'insieme singoletto, se ne dia la definizione e la si giustifichi ricorrendo all'uso di AF ed I definite nell'esercizio precedente.
  (c) Si fornisca la definizione delle funzioni min e max e anche in questo caso le si giustifichi come in (b).
  (d) Si mostri infine, la definizione dell'overriding di `union`.
- (a) Si dica se e come cambia la metodologia in Java per trattare i costruttori a nomi assegnati nel caso di Tipi di Dato e in quello e Tipi Astratti.
  (b) Si presenti la metodologia da utilizzare;
  (c) Si diano le ragioni per tale eventuale cambiamento ovvero le ragioni per non avere alcun cambiamento.
- Le eccezioni sono un meccanismo per fare "recovery" in caso di situazioni di comportamento anomalo ma previsto. Nondimeno il ricorso al sollevamento di eccezione deve essere attentamente considerato: Perchè?
- Il tipo di dato definito nella classe `OrdSet` è un Tipo Astratto o solo un Tipo di Dato ?
  (a) Si fornisca una risposta e si spieghi come in Java si possa dare risposta a tale quesito.
  (b) Si mostri una possibile definizione di `Set<A>`.
  (c) Si giustifichi la scelta fatta in (b) rispetto ad altre possibili.
- Si fornisca un Tipo Astratto per valori coppia polimorfi e modificabili. In particolare, si fornisca:
  (a) API.
  (b) ADT, stato concreto, AF ed I,
  (c) Implementazione e caso di uso.

Altri Esercizi in EserciziL15.pdf