Структури от Данни и Обектно-Ориентирано Програмиране
спец. Компютърни Науки
Упражнение №7
15.04.2010г.

ТЕМА: Параметризирани класове и интерфейси.
Структура от данни мултимножество

Родов тип – референтен тип, който има параметър-тип. Всеки параметър-тип се заменя с аргумент-тип когато се създава екземпляр на родовия тип.
Типът-параметър може да се използва като тип на нестатично поле, тип на параметър или връщана стойност на нестатичен метод, или локална променлива.

```
public class GenericClass<T> {
  T field;

  public T getField () {
    return field;
  }

  public void setField (T field) {
    this.field = field;
  }

  public void someOtherMethod () {
    T local = field;
    . . . . .
  }
}
```

Параметризиран тип – екземпляр на родов тип, в който типът-параметър е заместен с тип - актуален аргумент, например

GenericClass <String> str = new GenericClass <String>();

е конкретен параметризиран тип, получен при заместване на типа-параметър ***T*** с типа – актуален аргумент ***String*** и може да се използва подобно на другите референтни типове.

Задаване на граници (долна и горна) на типа-параметър:

```
class CMP<T extends Number & Comparable<T>> {
  CMP (T first, T second) {
    System.out.println (first.compareTo (second));
  }
}
```

Област на действие на типа-параметър е целият клас/интерфейс, в който е обявен и в нея той може да бъде припокрит:

```
class Outer<T> {
   class Inner<T extends Number> { }
}
```

T от класа ***Outer*** и ***T*** от класа ***Inner*** са различни! Пример:

```
Outer<String> str = new Outer<String> ();
Outer<String>.Inner<Float> i = str.new Inner<Float> ();
```

Типът – актуален аргумент може да бъде:

- конкретен тип – например:
  `List<Integer> list = new ArrayList<Integer>();`
- конкретен параметризиран тип – например:
  `List<List<Integer>> list = new ArrayList<List<Integer>> ();`
- тип масив – например:
  `List<int []> list = new ArrayList<int []> ();`
- жокер.

Всички типове с изключение на изброените типове, анонимните вътрешни класове и наследниците на ***Throwable*** могат да бъдат родови.

Всички параметризирани типове, които са екземпляри на един и същ родов тип делят един и същ клас по време на изпълнение, така нареченият „суров тип”.

```
Vector<String> x = new Vector<String>();
Vector<Integer> y = new Vector<Integer>();
boolean b = x.getClass() == y.getClass();  // b = true
```

Следният програмен фрагмент:

```
System.out.println("runtime type of ArrayList<String>: ”
      + newArrayList<String>().getClass());
System.out.println("runtime type of ArrayList<Long>  : "
      + new ArrayList<Long>().getClass());
```

Извежда:

```
runtime type of ArrayList<String> : class java.util. ArrayList
runtime type of ArrayList<Long>   : class java.util. ArrayList
```

Параметризираните типове, получени от един и същ родов тип, нямат връзка помежду си:

```
void printAll(ArrayList<Object> c) {
  for (Object o : c)
    System.out.println(o);
}
```

```
ArrayList<String> list = new ArrayList<String>();
... fill list ...
printAll(list);   // error
```

Грешката в горния пример произлиза от факта, че типът ArrayList<Object> не е суперклас на ArrayList<String>.

Не може да се създава обект от тип-параметър:

```
class Pair<E, T>  {
        E first;
        T second;
        Pair(E first, T second){
                this.first = first;
                this.second = second;
        }
        Pair(){
                this.first = new E();  // error
                this.second = new T();  // error
        }
        E getFirst() {
                return first;
        }
        T getSecond() {
                return second;
        }
        void setSecond(T s) {
                second = s;
        }
}
```

Може да се декларира тип-масив, чиито елементи са от параметризиран тип или променлива-тип, но не и такива масиви – обекти.

```
static void test() {
        Pair<Integer,Integer>[] intPairArr;  // деклариране
        //intPairArr = new Pair<Integer,Integer>[10] ; // error
        intPairArr = (Pair<Integer,Integer>[])new Object[10] ;  // създаване
        addElements(intPairArr);
        Pair<Integer,Integer> pair = intPairArr[1];
        Integer i = pair.getFirst();
        pair.setSecond(i);
}
```

Пример за деклариране на масив с елементи от „суров тип”:

```
static void test() {
  Pair[] intPairArr = new Pair[10] ;
```

```
  addElements(intPairArr);
  Pair<Integer,Integer> pair = intPairArr[1];  // unchecked warning
  Integer i = pair.getFirst();     // ClassClassException
  pair.setSecond(i);
}
```

Родов метод – съдържа параметър-тип преди типа на връщаната си стойност.

```
 public static <T> T max(T objA, T objB) {…}

public static <T extends Comparable<T>> T max(T objA, T objB) {
   if (objA.compareTo(objB) > 0)
     return objA;
   else
     return objB;
}
```

Пример за припокриване на методи:

```
static void overloadedMethod( Object o) {
      System.out.println("overloadedMethod(Object) called");
}
static void overloadedMethod( String s) {
      System.out.println("overloadedMethod(String) called");
}
static void overloadedMethod( Integer i) {
      System.out.println("overloadedMethod(Integer) called");
}

static <T> void genericMethod(T t) {
      overloadedMethod (t) ;  // кой метод се извиква?
}

public static void main(String[] args) {
      genericMethod( "abc" );
}
```

```
overloadedMethod(Object) called
```

Причината за този резултат е, че методът ***genericMethod*** след транслация изглежда така:

```
void genericMethod( Object t) {
      overloadedMethod (t) ;
}
```

Задача: Ненаредена конфигурация с повторение /**мултимножество**/ ще наричаме множество от наредени двойки от вида: $\{(e_1,n_1),(e_2,n_2),...,(e_k,n_k)\}$, където $e_1,e_2,...,e_k$ са два по два различни обекти – ще ги наричаме елементи на мултимножеството, а n_i са числа, които показват броя на повторенията /кратността / на $e_i, i=1,2,...,k$ в мултимножеството.

Да се напише на езика Java клас, който реализира интерфейса **MultySet** и представя мултимножество.

public interface *MultySet* extends *Cloneable*{

public boolean isEmpty() {…} - проверява дали мултимножеството е празно;

public boolean equals(MultySet other) {…} - проверява равенство на мултимножества;

public int contains(Object e) {…} - връща кратността на елемента **e,** ако принадлежи на мултимножеството, иначе връща нула;

public int size() {…} - връща броя на елементите на мултимножеството, всеки преброен толкова пъти, колкото е кратността му;

public void clear() {…} - унищожава всички елементи на мултимножеството;

public void add(Object e, int card) {…} - добавя към мултимножеството елемент **e** с кратност **card**;

public int remove (Object e) {…} - отстранява от мултимножеството елемента **e** и връща неговата кратност;

public MultySet clone() {…} – създава копие на обект-мултимножество;

public boolean subSetOf(MultySet other) {…} - проверява дали мултимножеството **this** е подмножество на **other**;

public MultySet union(MultySet other) {…} - намира обединението на мултимножествата **this** и **other;**

public MultySet intersection(MultySet other) {…} - намира сечението на мултимножествата **this** и **other;**

public MultySet difference(MultySet other) {…} - намира разликата на мултимножествата **this** и **other;**

public Iterator iterator() {…} - връща итератор за мултимножеството.

}

Класът да съдържа следните конструктори:

public MultySet() {…} - създава празно мултимножество;

public MultySet(Object [] array) {…} - създава мултимножество от елементите на масива array

```
import java.util.Iterator;

public interface MultySet<E> {
	public boolean isEmpty();
	public void clear();
	public Iterator<E> iterator();
	public int size();
	public MultySet<E> clone();
	public void add(E el, int card);
	public int remove(E el);
	public int contains(E el);
	public boolean equals(Object other);
	public String toString();
	public boolean SubSetOf(MultySet<E> other);
	public MultySet<E> union(MultySet<E> other);
	public MultySet<E> intersection(MultySet<E> other);
	public MultySet<E> difference(MultySet<E> other);
}

import java.util.*;

public class MultySetClass<E> implements MultySet<E> {

	public class Couple {
		private int card;
		private E element;

		public Couple(int c, E e) {
			card = c;
			element = e;
		}

		public boolean equals(Couple c) {
			return this.element.equals(c.element);
		}

		public String toString() {
			return "( " + element.toString() + " - " + card + " )";
		}
	}

	private class SetIterator implements Iterator<E> {
		int card = 0;
		E element = null;

		Iterator<Couple> it = body.iterator();
```

```
    public boolean hasNext()  {
       return card != 0 || it.hasNext();
    }

    public E next() {
        if(card > 0)
                card--;
        else {
                Couple c = it.next();
                card = c.card - 1;
                element = c.element;
        }
        return element;
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}

/* Vector body holds the elements of the multyset */
private Vector<Couple> body = new Vector<Couple>();

public MultySetClass() { }

public MultySetClass(E []array) {
        for(E a: array) {
                this.add(a, 1);
        }
}

private MultySetClass(Vector<Couple> v) {
        body = v;
}

public boolean isEmpty() {
        return body.isEmpty();
}

public void clear() {
        body.clear();
}

public Iterator<E> iterator() {
        return new SetIterator();
}
```

```
public Iterator<Couple> coupleIterator() {
	//return new CoupleIterator();
	return body.iterator();
}

public int size() {
	Iterator<Couple> it = coupleIterator();
	int count = 0;
	while(it.hasNext()){
		count += it.next().card;
	}
	return count;
}

public MultySetClass<E> clone() {
	MultySetClass<E> result = null;
	try {
		result = (MultySetClass<E>) super.clone();
	} catch (CloneNotSupportedException e) {
		e.printStackTrace();
	}
	result.body = (Vector<Couple>) body.clone();
	return result;
}

public void add(E el, int card) {
	Couple cNew = new Couple(card, el);
	Iterator<Couple> it = body.iterator();
	Couple c;
	int index = 0;
	while(it.hasNext()){
		c = it.next();
		if(c.equals(cNew)) {
			body.set(index, new Couple(cNew.card+c.card, c.element));
			return;
		}
		index++;
	}
	body.add(cNew);
}

public int remove(E el) {
	int result = 0;
	Iterator<Couple> it = body.iterator();
	Couple c = new Couple(1, el), tmp;
	while(it.hasNext()) {
		tmp = it.next();
		if(tmp.equals(c)) {
```

```
                    result = tmp.card;
                    it.remove();
                }
                break;
        }
        return result;
}

public int contains(E el){
        Iterator<Couple> it = coupleIterator();
        Couple c = new Couple(1, el), tmp;
        while(it.hasNext()){
                tmp = it.next();
                if(c.equals(tmp))
                        return tmp.card;
        }
        return 0;
}

public boolean equals(Object other) {
        if(this == other)
                return true;
        if(!(other instanceof MultySetClass))
                return false;
        MultySetClass<E> mSet = (MultySetClass<E>)other;
        Iterator<Couple> it = coupleIterator();
        Couple c;
        while(it.hasNext()) {
                if(mSet.contains((c = it.next()).element) != c.card)
                        return false;
        }
        Iterator<Couple> it1 = mSet.coupleIterator();
        while(it1.hasNext()) {
                if(this.contains((c = it1.next()).element) != c.card)
                        return false;
        }
        return true;
}

public String toString() {
        if(isEmpty())
                return "EMPTY";
        StringBuilder buffer = new StringBuilder();
        Iterator<Couple> it = coupleIterator();
        Couple tmp;
        while(it.hasNext()) {
                tmp = it.next();
                buffer.append(tmp.toString());
```

```
            if(it.hasNext())
                buffer.append(" , ");
        }
        return buffer.toString();
}

public boolean SubSetOf(MultySet<E> other) {
        Iterator<Couple> it = coupleIterator();
        Couple c;
        while(it.hasNext()) {
            c = it.next();
            if(other.contains(c.element) == 0)
                return false;
        }
        return true;
}

public MultySet<E> union(MultySet<E> other) {
        MultySetClass<E> result = this.clone();
        Iterator<Couple> it = ((MultySetClass)other).coupleIterator();
        Couple c;
        while(it.hasNext()) {
            c = it.next();
            result.add(c.element, c.card);
        }
        return result;
}

public MultySet<E> intersection(MultySet<E> other) {
        MultySetClass<E> result = new MultySetClass<E>();
        Iterator<Couple> it = coupleIterator();
        Couple c;
        int n;
        while(it.hasNext()) {
            c = it.next();
            n = other.contains(c.element);
            if(n != 0)
                result.add(c.element, Math.min(c.card, n));
        }
        return result;
}

public MultySet<E> difference(MultySet<E> other) {
        MultySetClass<E> result = new MultySetClass<E>();
        Iterator<Couple> it = coupleIterator();
        Couple c;
        while(it.hasNext()) {
            c = it.next();
```

```
                if(other.contains(c.element) != 0)
                    result.add(c.element, c.card);
            }
            return result;
        }

}
```