Laborator Generice si Colectii in Java

Teme laborator colectii si generice

1. Considerand principalele interfete specifice si clasele asociate colectiilor fara generice. Dezvoltati aplicatii pentru *List, Set* si *Map*, (minim 3). Utilizati exemplele si documentele aditionale.

2. Considerand evolutia colectiilor ce permit si parametrii de tip generic dezvoltati aplicatii pornind de la exemplele date in care sa folositi mecanismul generic, (minim 3).

3. Implementati si afisati produsul scalar a doua liste de tip double *u* si *v* de *n* elemente, folosind colectii clasice (List si/sau tablou double) si generice.

4. Verificati/testati/extindeti elementele specifice programarii in Java cu generice (folositi exemplele de la curs):

- un tip generic

-doua tipuri generice

-specializari (bounds)

-wild card, ?

-wild card specializat (upper/lower bounded)

-metode generice in clase generice

-interfete generice

-raw type- clasa generica fara parametrii

-ierarhii de clase generice

1. Generics general

T is a type variable or formal type parameter, when we use the class we have to specify what T is:

```
Box<Integer> integerBox;
```

Here <Integer> is a type argument which tells the compiler what kind of data we will place in the Box.

In order to instantiate an object, we use the keyword new like usual but also specifying the type argument:

```
Box<Integer> integerBox = new Box<Integer>();
```

And now we see the class in use, it is important to notice we don't have to use a cast operator, the compiler knows we are passing and receiving integers:

```
class Box<T> {
    private T t;
    public void add(T t) {
        this.t = t;
    }
    public T get() {
        return t;
    }
}

class BoxDemo {
    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        integerBox.add(new Integer(10));
        Integer someInteger = integerBox.get(); // no cast!
        System.out.println(someInteger);
    }
}
```

2. Generics List

This example illustrates how using a List (which normally accepts any data type) with the formal type parameter String only allows the programmer to place Strings in the List. If we tried to place an integer in the list a compile time error would occur.

```
import java.util.*;

class GenLists {

	private void printList(List<String> l) {
		Iterator<String> i = l.iterator();
		while(i.hasNext()) {
			System.out.println("Item: "+i.next());
		}
	}

	public static void main(String argv[]) {
		GenLists e = new GenLists();

		List<String> list = new ArrayList<String>();
		list.add(new String("Hello world!"));
		list.add(new String("Good bye!"));

		e.printList(list);
	}
}
```

3. Bounds –Wildcards

Generic type parameters in Java are not limited to specific classes. Java allows the use of wildcards to specify bounds on the type of parameters a given generic object may have. Wildcards are type parameters of the form "?", possibly annotated with a bound. Given that the exact element type of an object with a wildcard is unknown, restrictions are placed on the type of methods that may be called on the object.

As an example of an unbounded wildcard, List<?> indicates a list which has an unknown object type. Methods which take such a list as an argument can take any type of list, regardless of parameter type. Reading from the list will return

objects of type Object, and writing non-null elements to the list is not allowed, since the parameter type is not known.

```
import java.util.*;


class Wildcards {

     public void processElements(List<?> elements){
          // for each object in the list
          for(Object o : elements){
               System.out.println(o);
          }
     }

     public static void main(String argsp[]){
         List<String> list = new ArrayList<String>();
         list.add(new String("Hello world!"));
         list.add(new String("Good bye!"));

         List<Integer> intlist = new ArrayList<Integer>();
         intlist.add(new Integer(2));
         intlist.add(new Integer(7));
         intlist.add(new Integer(-3));

         Wildcards wild = new Wildcards();

         // we call the same method with two types of lists
         wild.processElements(list);
         wild.processElements(intlist);
     }
}
```

4. Bounded-wildcards

To specify the upper bound of a generic element, the extends keyword is used, which indicates that the generic type is a subtype of the bounding class. Thus it must either extend the class, or implement the interface of the bounding class. So List<? extends Number> means that the given list contains objects of some unknown type which extends the Number class.

```java
import java.util.*;


class Bounded {


     public void processElements(List<? extends Number>
elements){
          double sum = 0.0;

          // for each object in the list
          for(Number o : elements){
               sum += o.doubleValue();
          }

          System.out.println(sum);


     }

     public static void main(String args[]){
         List<Integer> list = new ArrayList<Integer>();
         list.add(new Integer(2));
         list.add(new Integer(7));
         list.add(new Integer(-3));

         List<Float> listf = new ArrayList<Float>();
         listf.add(new Float(1.1));
         listf.add(new Float(9.3));
         listf.add(new Float(-3.2));

         List<String> lists = new ArrayList<String>();
         lists.add(new String("Hello world!"));
         lists.add(new String("Good bye!"));

         Bounded wild = new Bounded();

         // we call the same method with two types of lists
         wild.processElements(list);
         wild.processElements(listf);
         //wild.processElements(lists);  // String does not
extend Number
     }
}
```

The use of wildcards above is necessary since objects of one type parameter cannot be converted to objects of another parameter. Neither List<Number> nor List<Integer> is a subtype of the other, even though Integer is a subtype of Number. So, code that deals with List<Number> does not work with List<Integer>. If it did, it would be possible to insert a Number that is not a Integer into it, which violates type safety.

It's also possible to specify a lower bound by using the super keyword instead of extends. The code <? super Number>, therefore, would be read as "an unknown type that is a supertype of Number, possibly Number itself". This variant is used less often.

5. Generics methods

It is possible to generify methods in Java. This method specifies a type T that is used as both type for the element parameter and the generic type of the List. We can call the addAndReturn() method using both String's and Integer's and their corresponding collections. The compiler knows from the type of the T parameter and collection<T> parameter definitions that the type is to be taken from these parameters at call time (use time).

```
import java.util.*;

class Methods {
     public static <T> T addAndReturn(T element, List<T> list){
         list.add(element);
         return element;
     }

     public static void main(String args[]){
          String str = "A String";
          List<String> stringList = new ArrayList<String>();

          String thestr = addAndReturn(str, stringList);

          Integer anint = new Integer(123);
          List<Integer> integerList = new ArrayList<Integer>();

          Integer theint = addAndReturn(anint, integerList);
     }
}
```

6. Generic Collections

As explained above the main reason for generics are collections. Used in previous examples, a List is a subtype of a Collection. In this example, we use a Set and a Map. The set has the property that an element cannot exist twice and a Map assigns a relation between and element and a key. In addition, two specific iteration methods are evidentiated: using the iterator and using a special for loop. The iterator is and interface implemented in Collection class but can also be implemented by programmers in their own classes. The new for loop means Object o takes one value of the collection each step, it iterates on its own.

```
import java.util.*;

class ExCollections {

     void dispIterator(Collection<?> c){
          Iterator<?> i = c.iterator();
          while(i.hasNext())
               System.out.println(i.next());
     }

     void dispFor(Collection<?> c){
          for(Object o : c)
               System.out.println(o);
     }


          public static void main(String argsp[]){
          ExCollections example = new ExCollections();

          Set<String> set = new HashSet<String>();
          set.add("a string");
          set.add("another string");
          set.add("last string");
          example.dispIterator(set);
          example.dispFor(set);

     Map<Integer, String> map = new HashMap<Integer, String>();

          map.put(1, "uno");
          map.put(2, "twin");
          map.put(300, "Sparta");
          map.put(20, "round");
```

```
            example.dispIterator(map.keySet());
            example.dispFor(map.values());


        }
}
```