Структури от Данни и Обектно-Ориентирано Програмиране
спец. Компютърни Науки
Упражнение №6
07.04.2010г.

ТЕМА: Вложени класове и интерфейси. Клас *java.util.Vector*
Структура от данни ОПАШКА

Деклариране на вложен клас:

```
class EnclosingClass {
   ...
   class NestedClass {
      ...
   }
}
```

Статичен вложен клас – има статут на член на класа, който се отразява на правото му на достъп до членове на класа и обекта. Вложените интерфейси са по премълчаване статични.

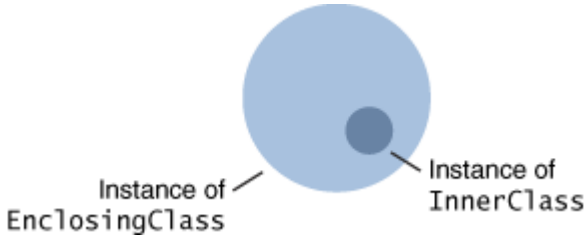
```
class EnclosingClass {
   ...
   static class StaticNestedClass {
      ...
   }
}
```

За да се създаде и да съществува обект от статичния вложен клас не е необходимо да съществува обект от външния клас. Аналогично, създаването на обект от външния клас не води автоматично до създаване на обект от вложения клас.
Методите на вложения клас, независимо дали са статични или не, имат достъп ***само до статичните членове*** на външния клас.

```
public class R {
  int y = 1;
  static int z = 2;

   public static class S {
      int x;

      void f () {
            System.out.println("f(): x=" + x);
            //System.out.println("f(): y=" + y); //Static reference to a non-static field
            System.out.println("f(): z=" + z);
      }
```

```
        static void g() {
            //System.out.println("g(): x=" + x);  //Static reference to a
                                                  // non-static  field
            // System.out.println("g(): y=" + y); //Static reference to a
                                                  // non-static  field
            System.out.println("g(): z=" + z);
        }
    } // end of class S
}  // end of class R

public class UseR {

  public static void main(String[] args) {
      R.S obj = new R.S();
      obj.f();
      R.S.g();
      obj.g();  // Warning
  }
}
```

Нестатичен вложен клас – вътрешен клас. Не може да декларира статични членове, с изключение на константи. Обект на вътрешния клас съществува само в обект на обхващащия клас и има достъп до неговите членове. Обект от външния клас може да бъде асоцииран с повече от един обект от вътрешния клас.



Тъй като всеки екземпляр на вътрешния клас е вложен в екземпляр на външния, то неговите методи имат достъп до всички членове - полета и методи на външния.

```
public class A {
  int y = 1;
  static int z = 2;

  public void useB() {
            B b = new B();
            b.f();
  }

  void h() {
            System.out.println("Outer metod called from inner");
```

```
    }


    public class B {  // innner class
            int x;
            public void f () {
                    System.out.println("f(): x=" + x);
                    System.out.println("f(): y=" + y);
                    System.out.println("f(): z=" + z);
                    h();
            }
            //static void g() {}  // cannot be declared static
    }
}

public class UseA {
    public static void main(String[] args) {
            //A.B obj = new A.B();  // no enclosing instance of A accessible
            A obj=new A();
            obj.useB();
            obj.new B().f();
    }
}
```

Задача: Да се състави клас на Java, който представя полиноми на една променлива с коефициенти от числов тип. Класът да бъде снабден с необходимите конструктори и методи за:

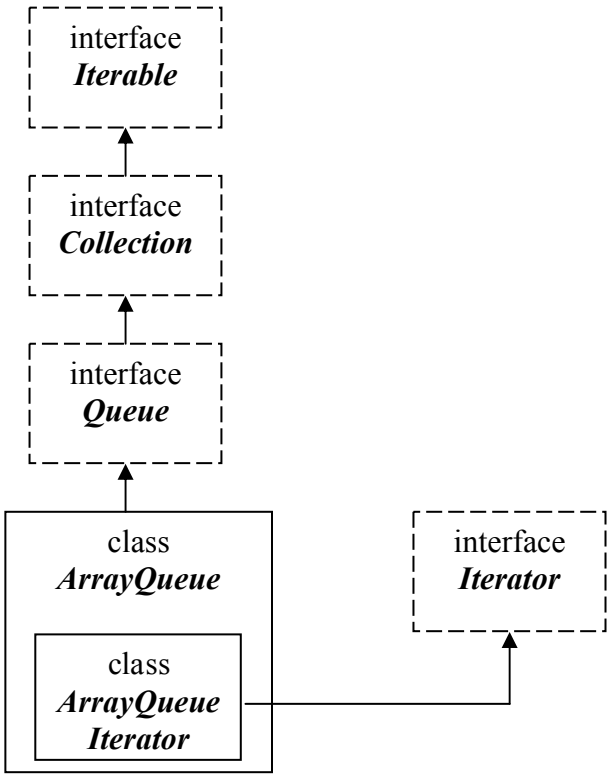
- събиране на полиноми;
- умножение на полиноми;
- пресмятане на стойността на полинома за дадена стойност на променливата;
- изчисляване на първа производна на даден полином.

Забележка: Коефициентите на полинома да се съхраняват в данна от тип java.util.Vector

```
import java.util.Vector;
public class Polinom implements Cloneable   {

    private int degree;
    private Vector<Number> coef;

    public Polinom (Number [] array) {
            degree = array.length-1;
            coef = new Vector<Number>(array.length);
            for(int i=0; i<array.length; i++)
                    coef.add(array[i]);
```

```
}

public Polinom (Polinom p) {  // Copy constructor
        degree = p.degree;
        coef = (Vector<Number>)p.coef.clone();
}

public Polinom (Vector<Number> v) {
        degree = v.size();
        coef = (Vector<Number>)v.clone();
}

private Polinom (int degree) {
        this.degree = degree;
        coef = new Vector<Number>(degree+1);
        for(int i=0; i<=degree; i++)
                coef.add(0);
}

public Number value(Number x) {
        double result = coef.elementAt(degree).doubleValue();
        double dx = x.doubleValue();
        for(int i=degree-1; i>=0; i--) {
                result = result * dx + coef.elementAt(i).doubleValue();
        }
        return new Double(result);
}

public Polinom add(Polinom other) {
        Polinom result;
        Polinom second;
        if(this.degree >= other.degree) {
                result = new Polinom(this);
                second = other;
        }
        else {
                result = new Polinom(other);
                second = this;
        }
        double d;
        for(int i=0; i<=second.degree; i++) {
                d = result.coef.elementAt(i).doubleValue() +
                        second.coef.elementAt(i).doubleValue();
                coef.set(i, d);
        }
        return result;
}
```

```
	public Polinom mult(Polinom other) {
		Polinom result = new Polinom(this.degree+other.degree);
		double d;
		for(int i=0; i<=this.degree; i++)
			for(int j=0; j<=other.degree; j++) {
				d = result.coef.elementAt(i+j).doubleValue() +
					this.coef.elementAt(i).doubleValue() *
					other.coef.elementAt(j).doubleValue();
				result.coef.setElementAt(d, i+j);
			}
		return result;
	}

	public boolean equals(Object o) {
		if(o == null)
			return false;
		if(!(o instanceof Polinom))
			return false;
		if(o == this)
			return true;
		Polinom tmp = (Polinom)o;
		return degree == tmp.degree && coef.equals(tmp.coef);
	}

	public Object clone() {
		Polinom result = null;
		try {
			result = (Polinom) super.clone();
		} catch (CloneNotSupportedException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
		result.coef = (Vector<Number>) coef.clone();
		return result;
	}

	public String toString() {
		StringBuffer str = new StringBuffer("");
		for(int i=degree; i>0; i--) {
			str.append(coef.elementAt(i) + "*x^" + i + " + ");
		}
		str.append(coef.elementAt(0));
		return str.toString();
	}
}
```




```
/**
 * This interface is used to indicate that a given class can be
 * iterated over.  The compiler uses this interface to determine which
 * classes are suitable targets of the foreach construct.
 */
public interface Iterable<E> {

        /**
         * Returns an iterator for the collection.
         */
        Iterator<E> iterator ();
}


public interface Collection<E> extends Iterable<E> {

        /*
         * Ensures that this collection contains the specified element (optional operation).
         * Returns true if this collection changed as a result of the call.
         * Returns false if this collection does not permit duplicates and already
         * contains the specified element.
         */
        boolean add(E e);  //throws IllegalStateException, NullPointerException,
                                                //UnsupportedOperationException;
```

```
/*
 * Returns true if this collection contains the specified element.
 *  More formally, returns true if and only if this collection contains at least
 *  one element e such that (o==null ? e==null : o.equals(e)).
 */
boolean contains(Object o);

/*
 * Returns an iterator over the elements in this collection.
 * There are no guarantees concerning the order in which the elements
 * are returned (unless this collection is an instance
 * of some class that provides a guarantee).
 */
Iterator<E> iterator();

/*
 * Removes a single instance of the specified element from this collection,
 * if it is present (optional operation). More formally, removes an element e such that
 * (o==null ? e==null : o.equals(e)),
 * if this collection contains one or more such elements.
 * Returns true if this collection contained the specified element (or equivalently,
 * if this collection changed as a result of the call).
 */
boolean remove(Object e);  //throws NoSuchElementException,
                           //UnsupportedOperationException

/*
 * Returns the number of elements in this collection.
 * If this collection contains more than Integer.MAX_VALUE elements,
 * returns Integer.MAX_VALUE.
 */
 int size();

/*
 * Returns true if this collection contains no elements
 */
boolean isEmpty();

/*
 * Removes all of the elements from this collection (optional operation).
 * The collection will be empty after this method returns
 */
void clear();  //throws UnsupportedOperationException
```

```
    /*
     * Returns an array containing all of the elements in this collection;
     * the runtime type of the returned array is that of the specified array.
     * If the collection fits in the specified array, it is returned therein.
     * Otherwise, a new array is allocated with the runtime type of the specified array
     * and the size of this collection.
     * If this collection fits in the specified array with room to spare
     * (i.e., the array has more elements than this collection),
     * the element in the array immediately following the end of the collection
     * is set to null. (This is useful in determining the length of this collection
     * only if the caller knows that this collection does not contain any null elements.)
     */
    <T> T[] toArray(T[] a);  //throws UnsupportedOperationException
}

public interface Queue<E> extends Collection<E>{

    /*
    * Inserts the specified element into this queue if it is possible to do so immediately
    * without violating capacity restrictions, returning true upon success
    * and throwing an IllegalStateException if no space is currently available.
    */
    boolean add(E e);  //throws IllegalStateException, NullPointerException

    /*
     * Inserts the specified element into this queue if it is possible to do so immediately
     * without violating capacity restrictions. When using a capacity-restricted queue,
     * this method is generally preferable to add(E), which can fail to insert an element
     * only by throwing an exception.
     */
    boolean offer(E e);  //throws NullPointerException

    /*
     * Retrieves and removes the head of this queue.
     * This method differs from poll only in that it throws an exception
     *  if this queue is empty.
     */
    E remove(); //throws NoSuchElementException

    /*
     * Retrieves and removes the head of this queue, or returns null if queue is empty
     */
    E poll();
```

```
        /*
         * Retrieves, but does not remove, the head of this queue.
         * This method differs from peek only in that it throws an exception
         *  if this queue is empty.
         */
        E element();  //throws NoSuchElementException

        /*
         * Retrieves, but does not remove, the head of this queue,
         * or returns null if queue is empty.
         */
        E peek();
}


public interface Iterator<E> {

        /*
         * Returns true if the iteration has more elements.
         * In other words, returns true if next would return
         * an element rather than throwing an exception.
         */
        boolean hasNext();

        /*
         * Returns the next element in the iteration.
         */
        E next();

        /*
         * Removes from the underlying collection the last element
         * returned by the iterator (optional operation).
         * This method can be called only once per call to next.
         * The behavior of an iterator is unspecified if the underlying collection
         * is modified while the iteration is in progress
         * in any way other than by calling this method.
         */
        void remove();
}
```

```
import java.util.NoSuchElementException;

public class ArrayQueue<E> implements Queue<E> {

    private final E[] items;
    private int head;
    private int tail;
    private int size;

    public ArrayQueue(int capacity) {
        items = (E[])new Object[capacity];
    }

    final int inc(int i) {
        return (++i == items.length)? 0 : i;
    }

    public boolean add(E e) {  //throws IllegalStateException, NullPointerException
        throw new UnsupportedOperationException();
    }

    public boolean offer(E e) {  //throws NullPointerException
        if(size == items.length)
            return false;
        items[tail] = e;
        tail = inc(tail);
        size++;
        return true;
    }

    public E remove() { //throws NoSuchElementException
        throw new UnsupportedOperationException();
    }

    public E poll() {
        if(size == 0)
            return null;
        E result = items[head];
        items[head] = null;
        head = inc(head);
        size--;
        return result;
    }

    public E element() {  //throws NoSuchElementException
        throw new UnsupportedOperationException();
    }
```

```
public E peek() {
        if(size == 0)
                return null;
        return items[head];
}

public boolean contains(Object o) {
        throw new UnsupportedOperationException();
}

public boolean remove(Object e) {  //throws NoSuchElementException,
                                   // UnsupportedOperationException
        throw new UnsupportedOperationException();
}

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

public int size() {
        return size;
}

public boolean isEmpty() {
        return size == 0;
}

public void clear() {  //throws UnsupportedOperationException
        throw new UnsupportedOperationException();
}

public String toString() {
        StringBuffer stb = new StringBuffer("[ ");
        if(size > 0) {
                Iterator<E> it = iterator();
                while(it.hasNext())
                        stb.append(it.next().toString() + " ; ");

                stb.delete(stb.length()-2, stb.length()-1);
                stb.append("]");
        }
        return stb.toString();
}

public <T> T[] toArray(T[] a) {
        throw new UnsupportedOperationException();
}
```

```
  /**
   * Iterator for ArrayQueue
   */
  private class ArrayQueueIterator implements Iterator<E> {

  /**
  * Index of element to be returned by next,
  * or a negative number if no such.
  */
  private int nextIndex;

/**
 * nextItem holds on to item fields because once we claim
 * that an element exists in hasNext(), we must return it in
 * the following next() call even if it was in the process of
 * being removed when hasNext() was called.
 */
private E nextItem;

/**
 * Index of element returned by most recent call to next.
 * Reset to -1 if this element is deleted by a call to remove.
 */
private int lastRet;

ArrayQueueIterator() {
    lastRet = -1;
    if (size == 0)
        nextIndex = -1;
    else {
        nextIndex = head;
        nextItem = items[head];
    }
}

public boolean hasNext() {
    return nextIndex >= 0;
}

/**
 * Checks whether nextIndex is valid; if so setting nextItem.
 * Stops iterator when either hits tail or sees null item.
 */
private void checkNext() {
    if (nextIndex == head) {
        nextIndex = -1;
        nextItem = null;
    }
```

```
            else {
                nextItem = items[nextIndex];
                if (nextItem == null)
                    nextIndex = -1;
            }
        }

        public E next() {
          if (nextIndex < 0)
                throw new NoSuchElementException();
            lastRet = nextIndex;
            E e = nextItem;
            nextIndex = inc(nextIndex);
            checkNext();
            return e;
        }

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