Структури от Данни и Обектно-Ориентирано Програмиране
спец. Компютърни Науки
Упражнение №1
24.02.2009г.

ТЕМА: Преговор. Наследяване и полиморфизъм

Писмен изпит по Увод в Програмирането

Задача 1: Отсечка наричаме двойката: начало и край – реални числа, като началото е по-малко или равно на края. Да се напише клас на езика Java за работа с отсечки, съдържащ:

1. Декларации на променливи за представяне на отсечка
2. Конструктор с два параметъра за инициализация на отсечка
3. Методи, реализиращи следните операции над отсечки:
   - Дължина на отсечка O(H,K): d(O) = K-H
   - Сума на две отсечки $O_1 + O_2 = (0, d(O_1) + d(O_2))$
   - Произведение на отсечка с реално число O * a = (0, d(O) *a)
   - Сравнение на две отсечки $O_1 \leq O_2$, ако $d(O_1) \leq d(O_2)$
4. Метод, който сортира масив от отсечки във възходящ ред

Забележка - Да се напише инвариант на класа.

```
public class Segment {

    private double begin, end;

    public Segment(double b, double e) {
        if(b > e)
            throw new IllegalArgumentException("Begin should be less or equal to
            the end!");
        begin = b;
        end = e;
     }

    public double length() {
        return end - begin;
    }

    public Segment sum(Segment o) {
        return new Segment(0, this.length() + o.length());
    }

    public Segment mult(double a) {
        return new Segment(0, a * this.length());
```

```
    }

    public boolean lessThan(Segment o) {
        return this.length() < o.length();
    }

    public boolean lessOrEqual(Segment o) {
        return this.length() <= o.length();
    }

    public String toString() {
        return "(" + begin + ", " + end + ')';
    }

    public static void sort(Segment [] array) {
        qSort(0, array.length-1, array);
    }

    static void qSort(int left, int right, Segment[] array) {
        int pivotIndex = -1;
        if(right > left) { // more than 1 element
            pivotIndex=partition(left, right, array);
        qSort(left, pivotIndex-1, array);
        qSort(pivotIndex+1, right, array);
        }
    }

static int partition(int left, int right, Segment [] array)  {
    int leftPtr=left;
    int rightPtr=right;
    if (!array[leftPtr].lessOrEqual(array[rightPtr]))
        swap(leftPtr, rightPtr, array);
    Segment pivot=array[left];
    do {
        do  leftPtr++;
        while (array[leftPtr].lessThan(pivot));
        do  rightPtr--;
        while (!array[rightPtr].lessOrEqual(pivot));
        if (leftPtr<rightPtr)
          swap(leftPtr, rightPtr, array);
    } while (leftPtr<rightPtr);
    int pivotIndex=rightPtr;
    swap(left, pivotIndex, array);
    return pivotIndex;
}
```

```
static void swap(int left, int right, Segment[] array) {
        Segment temp=array[left];
        array[left]=array[right];
        array[right]=temp;
    }
}
```

Задача 2: Правоъгълна мрежа се състои от проходими и непроходими квадратчета.От проходимо квадратче може да се премине във всяко от четирите му съседни, като се прекоси общата им страна. От непроходимо квадратче не може да се премине към съседно. Всяко проходимо квадратче по контура на мрежата е изход от мрежата. Да се напише клас на езика Java, съдържащ следните методи:

- Метод, който проверява дали има път между две квадратчета в мрежата
- Метод, който определя броят на изходите от мрежата за дадено квадратче

```
import java.util.Scanner;

public class Maze {

        boolean [][] maze;

        Maze(int n) {
                maze=new boolean[n][n];
        }

        void getMaze() {
           Scanner sc=new Scanner(System.in);
           for (int i=0; i<maze.length; i++)
             for (int j=0; j<maze.length; j++)
               maze[i][j]=sc.nextInt()!=0;
        }

        boolean isInMaze(int x, int y) {
               return x>=0 && x<maze.length &&
                           y>=0 && y<maze[0].length;
        }

        boolean isFree(int x, int y) {
                return maze[x][y];
        }
```

```
	boolean isExit(int x, int y) {
		return x==0 || x==maze.length-1 ||
				y==0 || y==maze[0].length-1;
	}

	void fill(int x, int y) {
	  maze[x][y]=false;
	}

	void undo(int x, int y) {
	  maze[x][y]=true;
	}
}

import java.util.Scanner;

public class FindAllExits {

	static Maze maze;

	public static void main(String[] args) {
		Scanner sc=new Scanner(System.in);
		System.out.print("Enter the size of maze: ");
		int n=sc.nextInt();
		maze=new Maze(n);
		maze.getMaze();
		int bX, bY;
		System.out.print("BEGIN X: ");
		bX=sc.nextInt();
		System.out.print("BEGIN Y: ");
		bY=sc.nextInt();
		inspectCell(bX, bY);
	}

	static void inspectCell(int x, int y) {
		if(maze.isInMaze(x, y) && maze.isFree(x, y)) {
			if(maze.isExit(x, y))
				register(x, y);
			maze.fill(x, y);
			findExits(x, y);
		}
	}
```

```
    static void findExits(int bX, int bY) {
        inspectCell(bX+1, bY);
        inspectCell(bX, bY+1);
        inspectCell(bX-1, bY);
        inspectCell(bX, bY-1);
    }

    static void register(int x, int y) {
       System.out.println("Cell(" + x + ',' + y + ") is an exit");
    }
}
```

Какво може да се прави в наследника на даден клас

- Да се използва наследено поле;
- Да се декларира поле с името на поле от базовия клас, при което последното се ***скрива***;
- Да се декларират нови полета;
- Да се ползват наследените методи;
- Да се създава нов метод на обекта, който има сигнатурата на метод от базовия клас, при което последният се ***припокрива***. При това типовете на връщаните стойности трябва да съвпадат;
- Да се създава нов метод на класа, който има сигнатурата на метод от базовия клас, при което последния се ***скрива***;
- Да се декларират нови методи;

Обекти. Претоварване, припокриване и скриване

Пример 1:

```
class Point {
        int x = 0, y = 0;
        void move(int dx, int dy) { x += dx; y += dy; }
        int color;
}

class RealPoint extends Point {
        float x = 0.0f, y = 0.0f;
        void move(int dx, int dy) { move((float)dx, (float)dy); }
        void move(float dx, float dy) { x += dx; y += dy; }
}
```

Пример 2: Грешка при наследяване

```
class Point {
        int x = 0, y = 0, color;
        void move(int dx, int dy) { x += dx; y += dy; }
        int getX() { return x; }
        int getY() { return y; }
}

class RealPoint extends Point {
        float x = 0.0f, y = 0.0f;
        void move(int dx, int dy) { move((float)dx, (float)dy); }
        void move(float dx, float dy) { x += dx; y += dy; }
        float getX() { return x; }
        float getY() { return y; }
}
```

Ред на изпълнение на конструкторите при наследяване

```
class Meal {
  Meal() { System.out.println("Meal()"); }
}

class Bread {
  Bread() { System.out.println("Bread()"); }
}

class Cheese {
  Cheese() { System.out.println("Cheese()"); }
}

class Lettuce {
  Lettuce() { System.out.println("Lettuce()"); }
}

class Lunch extends Meal {
  Lunch() { System.out.println("Lunch()");}
}

class PortableLunch extends Lunch {
  PortableLunch() {
    System.out.println("PortableLunch()");
  }
}
```

```
class Sandwich extends PortableLunch {
  Bread b = new Bread();
  Cheese c = new Cheese();
  Lettuce l = new Lettuce();
  Sandwich() {
    System.out.println("Sandwich()");
  }
  public static void main(String[] args) {
    new Sandwich();
  }
}
```

```
Meal()
Lunch()
PortableLunch()
Bread()
Cheese()
Lettuce()
Sandwich()
```

Полиморфно поведение на методи на обекта

```
class Actor {
  public void act() {}
}

class HappyActor extends Actor {
  public void act() { System.out.println("HappyActor"); }
}

class SadActor extends Actor {
  public void act() {  System.out.println("SadActor"); }
}

class Stage {
  private Actor actor = new HappyActor();
  public void changeRole() { actor = new SadActor(); }
  public void performPlay() { actor.act(); }
}
```

```
public class Transformation {
  public static void main(String[] args) {
    Stage stage = new Stage();
    stage.performPlay();
    stage.changeRole();
    stage.performPlay();
  }
}
```

Наследяване на полета и методи. Полиморфизъм

```
public class BaseClass {

        int x;
        int z;

        public BaseClass() {}

        public void method1() {
          System.out.println("Base method1");
        }

        public int x() {
                return x;
        }

        public BaseClass method4() {  // ???
                return new BaseClass();
        }
}

public class Derived extends BaseClass {
        int x=1;
        int y=2;
        float z=3;

        public Derived(){}

        public int x() {return x;}

        public void method1() {
                 System.out.println("Derived method1");
        }
```

```
        /* Overrides method with different return type
        public int method1() {
          System.out.println("Derived method1");
          return 1;
        }
        */

        public void method2() {
          System.out.println("Derived method2");
        }

        public void method3() {
          System.out.println(x);
          System.out.println(super.x);
        }

        public Derived method4() {
                return new Derived();
        }
}

public class Main {

        public static void main(String[] args) {
          BaseClass [] x= { new BaseClass(),  new Derived() };
          x[0].method1();  // Base method1
          x[1].method1();  // Derived Method1
          // x[0].method2();  // Compile Time Error
          // ((Derived)x[0]).method2();  // Class Cast Exception
          // x[1].method2();  // Compile Time Error
          ((Derived)x[1]).method2();
          ((Derived)x[1]).method3();
          System.out.println(x[0].x+" "+x[1].x+" "+((Derived)x[1]).y);  // 0 0 2
          System.out.println(x[0].x+" "+((Derived)x[1]).x);  // 0 1
          System.out.println(x[0].x()+" "+x[1].x()+" "+((Derived)x[1]).y);  // 0 1 2
          System.out.println(x[0].z+" "+x[1].z+" "+((Derived)x[1]).z);  // 0 0 3.0
        }
}
```

Скриване на статични методи

```
public class BaseClass {

        public static String staticMethod() {
           return "Base class staticMethod()";
        }

        public String dynamicMethod() {
           return "Base class dynamicMethod()";
        }

        public String dynamicMethod1() {
           return "Base class dynamicMethod1()";
        }
}

public class DerivedClass extends BaseClass {
        public static String staticMethod() {
           return "Derived class staticMethod()";
        }

        public String dynamicMethod () {
           return "Derived class dynamicMethod()";
        }
}

public class Main {

        public static void main(String[] args) {
                BaseClass base = new DerivedClass(); // Upcast
                System.out.println(base.staticMethod());
                System.out.println(base.dynamicMethod());
                System.out.println(base.dynamicMethod1());
        }
 }
```

Абстрактни класове

```
public abstract class Question {
        protected String theText;
        public Question() {
                theText = "";
        }

        public Question( String text ) {
                theText = text;
        }

        public abstract void askTheUser();
}

public class YesNoQuestion extends Question {
        public YesNoQuestion( String text ) {
                super( text );
        }

        public void askTheUser() {
                System.out.println( theText );
                System.out.println( "YES or NO ...?" );
        }
}


public class FreeTextQuestion extends Question {
        public FreeTextQuestion( String text ) {
                super( text );
        }

        public void askTheUser() {
                System.out.println( theText );
                System.out.println( "Well...?  What's the answer...?" );
        }
}

public class QuestionTest {
        public static void main(String[] args) {
                Question[] questions = getQuestions();
                for( int i = 0; i < questions.length; i++ ) {
                        questions[ i ].askTheUser();  // Polymorphism !!!
                }
        }
```

```
    private static Question[] getQuestions() {
        Question[] qs = new Question[ 2 ];
        qs[0] = new YesNoQuestion( "Do you understand polymorphism?" );
        qs[1] = new FreeTextQuestion( "Why is polymorphism good?" );
        return qs;
    }
}
```

Методи, общи за всички обекти (методи на класа Object)

1. Методи, които подлежат на припокриване: ***equals, hashCode, toString, clone***
   - public boolean **equals**(Object *obj*)

   Контракт на метода: реализира релация на еквивалентност – рефлексивна, симетрична, транзитивна.
   Непротиворечивост на реализацията.
   Сравнението с ***null*** трябва винаги да дава резултат ***false***.
   Кога да се припокрива **equals** – ако равенство на обекти надхвърля рамките на идентичност на обекти.

```
public boolean equals(Object obj)  {
        if(obj==null)
                return false;
        if(this==obj)
                return true;
        if(!(obj instanceof MyClass))
                return false;
        MyClass tmp=(MyClass)obj;
        // all necessary comparions
        return …;
}
```

Допустим начин за проверка на равенство, но не се препоръчва.

```
public boolean equals(MyObject obj)  {
        // all necessary comparions
        return …;
}
```

   - public int **hashCode**()

   Контракт на метода: ако два обекта са равни съгласно **equals** , то за тях методът **hashCode** връща една и съща стойност.
   Непротиворечивост на реализацията.
   Задължително припокриване в клас, който припокрива **equals.**

   - public String **toString**()

   Задължително припокриване.

getClass().getName() + '@' + Integer.toHexString(hashCode())

- public Object **clone**() throws CloneNotSupportedException

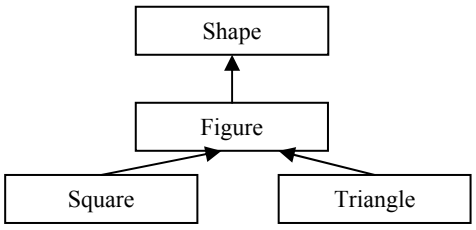
Контракт на метода: класът трябва да реализира интерфейса ***Cloneable*** и методът да изпълнява следните условия:
x.clone() != x
x.clone().getClass() == x.getClass()
x.clone().equals(x)

```
public Object clone() {
        try {
                return super.clone();
        }
        catch(CloneNotSupportedException e) {
                throw new Error("Assertion failure");  // Can't happen
        }
}
```

!!! Внимание, ако класът съдържа рефрентни полета.

2. Методи, които не подлежат на припокриване: ***getClass, notify, notifyAll, wait***

<u>Задача:</u> Да се съставят класовете от следната йерархия, като класът в основата е абстрактен. Всеки обект на класа *Shape* има цвят – един от фиксиран набор цветове, а също и площ, която се пресмята по подходящ начин. Класът *Figure* също е абстрактен, като обектите от този клас имат и метод за пресмятане на периметъра. Класовете *Square* и *Triangle*, които представят конкретни равнинни фигури, използват обекти на класа *Point* за представяне на съответната фигура.



public class *Color* {

```
        private String name;

        private Color(String nm) { name = nm; }

        public String toString() { return name; }
```

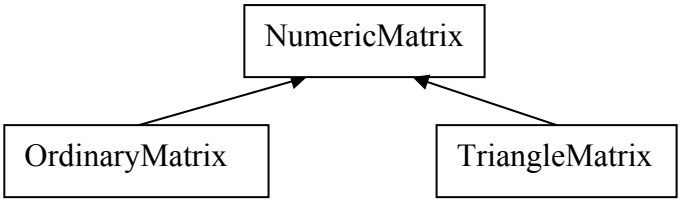
```
        public static final Color
           COLORLESS = new Color("COLORLESS"),
           WHITE = new Color("WHITE"),
           BLACK = new Color("BLACK"),
           BLUE = new Color("BLUE"),
           GREEN = new Color("GREEN"),
           RED = new Color("RED");
}

public class Point implements Cloneable {
        private double x=0;
        private double y=0;

        public Point()  {}


        public Point(double x, double y ) {
                this.x=x;
                this.y=y;
        }

        // Cloning constructor
        public Point(Point p) {
                this.x=p.x;
                this.y=p.y;
        }

        public double getX () {
                return x;
        }

        public double getY () {
                return y;
        }

        public boolean equals(Object p)  {
                if(p == null)
                        return false;
                if(this == p)
                        return true;
                if(!(p instanceof Point))
                        return false;
                Point tmp=(Point)p;
                return this.x==tmp.x && this.y==tmp.y;
        }
```

```
        // Possible but not recommended
        public boolean equals(Point p)  {
                return this.x==p.x && this.y==p.y;
        }

        public Object clone() {
                try {
                        return super.clone();
                }
                catch(CloneNotSupportedException e) {
                        throw new Error("Assertion failure");  // Can't happen
                }
        }

        public static Point newInstance(Point p) {
                return new Point(p);
        }

        public static boolean sameLine(Point a, Point b, Point c)  {
                throw new UnsupportedOperationException();
        }

        public double distanceTo(Point p) {
                throw new UnsupportedOperationException();
        }
}

public abstract class Shape {

        protected Color col=Color.COLORLESS;

        public Shape() {}

        public Color getColor() {
                return col;
        }

        public void setColor(Color c) {
                col=c;
        }

        abstract double area();
}
```

```
public class Square extends Figure {

  private Point leftUp;
  private Point rightDown;

  public Square(Point p1, Point p2) {
    this.leftUp=p1;
    this.rightDown=p2;
  }

  public boolean equals (Object q)  {
          throw new UnsupportedOperationException();
  }

  public double len()  {
          throw new UnsupportedOperationException();
  }

  public double diagonal()  {
          throw new UnsupportedOperationException();
  }

  public double perim() {
          throw new UnsupportedOperationException();
  }

  public double area()  {
          throw new UnsupportedOperationException();
  }
}

public class Triangle extends Figure {

  private Point a, b, c;

  public Triangle(Point a, Point b, Point c) throws InvalidFigure {
    if (Point.sameLine(a, b, c)) throw new InvalidFigure();
    this.a=a;
    this.b=b;
    this.c=c;
  }

  public Triangle(Point a, Point b, Point c, Color col) throws InvalidFigure {
    if (Point.sameLine(a, b, c)) throw new InvalidFigure();
    this.a=a;
    this.b=b;
```

```
    this.c=c;
    this.col=col;
  }

  private Triangle() {
    a=new Point(0,0);
    b=new Point(1,0);
    c=new Point(0,1);
  }

  public boolean equals(Object t) {
          throw new UnsupportedOperationException();
  }

  public double ab() {
          throw new UnsupportedOperationException();
  }

  public double ac() {
          throw new UnsupportedOperationException();
  }

  public double bc() {
          throw new UnsupportedOperationException();
  }

  public double area() {
          throw new UnsupportedOperationException();
  }

  public double perim () {
          throw new UnsupportedOperationException();
  }
}
```

Задача: Да се реализира следната йерархия от класове за представяне на числови матрици, като класът *NumericMatrix* е абстрактен. Елементите на матриците са тип *double*. Двата реални класа се различават по вида на матриците – правоъгълни или триъгълни, което се отразява в различното вътрешно представяне. Целта е двуаргументните операции с матрици, като събиране и умножение, да се извършват успешно независимо от вида на двата аргумента. Ако аргументите са от един вид, то резултатът е от същия вид, а ако са от различни видове – то резултатът е правоъгълна матрица.




```
public abstract class NumericMatrix {

    protected int rows;
    protected int columns;

    abstract double elementAt(int i, int j) throws IllegalArgumentException;

    abstract void setElement(int i,int j,double val) throws IllegalArgumentException;

    public abstract NumericMatrix copyMatrix();

    boolean canAdd(NumericMatrix mat) {
      return this.rows==mat.rows && this.columns==mat.columns;
    }

    boolean canMult(NumericMatrix mat) {
      return this.columns==mat.rows;
    }

    public NumericMatrix() {}

    public NumericMatrix addTo(NumericMatrix mat) throws
                    IllegalArgumentException {
        System.out.println("Here I am in Numeric!");
        if (!canAdd(mat)) throw new
                        IllegalArgumentException("Can't do addition!");
        NumericMatrix result=new OrdinaryMatrix(rows, columns);
        for (int i=0; i<rows; i++)
```

```
                for (int j=0; j<columns; j++)
                  try {
                    result.setElement(i,j, this.elementAt(i,j)+mat.elementAt(i,j));
                  }
                  catch(IllegalArgumentException e) {
                  System.out.println(e.toString());
                }
                return result;
        }

        public NumericMatrix multWith(NumericMatrix mat) throws
                                IllegalArgumentException {
                throw new UnsupportedOperationException();
        }

        public void printMatrix() {
                for(int i=0; i<rows; i++) {
                        try {
                                for(int j=0; j<columns-1; j++)
                                        System.out.print(elementAt(i, j) + " , ");
                                System.out.println(elementAt(i, columns-1));
                        }
                        catch(IllegalArgumentException e) {}
                }
        }
}

public class OrdinaryMatrix extends NumericMatrix {

        private double body [][];

        public OrdinaryMatrix(int r, int c) {
          rows=r;
          columns=c;
          body=new double[r][c];
        }

        public OrdinaryMatrix(double [][]array) {
          rows=array.length;
          columns=array[0].length;
          body=new double[rows][columns];
          for (int i=0; i<rows; i++)
            for (int j=0; j<columns; j++)
              body[i][j]=array[i][j];
        }
```

```
public NumericMatrix copyMatrix()  {
  return new OrdinaryMatrix(body);
}

public void setElement(int i, int j, double val)
                    throws IllegalArgumentException {
    if (i<0 || i>=rows || j<0 || j>=columns)
      throw new IllegalArgumentException("Index out of matrix range");
    body[i][j]=val;
}

public double elementAt(int i, int j)throws IllegalArgumentException  {
          if (i<0 || i>=rows || j<0 || j>=columns)
             throw new IllegalArgumentException("Index out of matrix range");
          return body[i][j];
}

public OrdinaryMatrix addTo(OrdinaryMatrix mat) throws
                    IllegalArgumentException {
          System.out.println("Here I am in Ordinary!");
          if (!canAdd(mat)) throw
                    new IllegalArgumentException("Can't do addition!");
          double temp[][]=new double[body.length][body[0].length];
          for (int i=0; i<rows; i++)
          for (int j=0; j<columns; j++)
                    temp[i][j]=body[i][j]+mat.body[i][j];
          return new OrdinaryMatrix(temp);
}

public NumericMatrix multWith(OrdinaryMatrix mat) throws
                    IllegalArgumentException {
          throw new UnsupportedOperationException();
}

// /*
public void printMatrix() {
          for(int i=0; i<rows; i++) {
                    for(int j=0; j<columns-1; j++)
                              System.out.print(body[i][j] + " , ");
                    System.out.println(body[i][columns-1]);
          }
}
// */

}
```

```
public class TriangleMatrix extends NumericMatrix {

        private double body[];
        private int numElem;

        public TriangleMatrix(int r) {
           rows=r;
           columns=r;
           numElem=(r+1)*r/2;
           body=new double[numElem];
        }

        public TriangleMatrix(double [][]array) {
           if(array.length!=array[0].length)
                    throw new IllegalArgumentException("Matrix is not triangle one");
           rows=columns=array.length;
           numElem=(array.length+1)*array.length/2;
           body=new double[numElem];
           int index=0;
           for (int i=0; i<rows; i++)
             for  (int j=i; j<columns; j++)
               body[index++]=array[i][j];
        }

        private TriangleMatrix (int r, double []array)  {
                  rows=columns=r;
                  numElem=array.length;
                  body=new double[numElem];
                  for (int i=0; i<numElem; i++)
                            body[i]=array[i];
        }

        private int indexOf(int i, int j)  {
                  if (j<i) return -1;
                  else return (2*columns-i+1)*i/2+j-i;
        }

        public NumericMatrix copyMatrix()  {
                  return new TriangleMatrix(rows, body);
        }

        public void setElement(int i, int j, double val)throws IllegalArgumentException {
                  if (i<0 || i>=rows || j<0 || j>=columns)
                    throw new IllegalArgumentException("Index out of matrix range");
                  int temp=indexOf(i,j);
```

```
            if(temp!=-1)
                body[temp]=val;
    }

    public double elementAt(int i, int j) throws IllegalArgumentException {
            if (i<0 || i>=rows || j<0 || j>=columns)
              throw new IllegalArgumentException("Index out of matrix range");
            int temp=indexOf(i,j);
            if(temp==-1) return 0;
            else return body[temp];
    }

    public TriangleMatrix addTo(TriangleMatrix mat) throws
                IllegalArgumentException {
            System.out.println("Here I am in Triangle!");
            if (!canAdd(mat)) throw new
                IllegalArgumentException("Can't do addition!");
            TriangleMatrix result=(TriangleMatrix)copyMatrix();
            for (int i=0; i<numElem; i++)
                result.body[i]+=mat.body[i];
            return result;
    }

    public NumericMatrix multWith(TriangleMatrix mat) throws
                IllegalArgumentException {
            throw new UnsupportedOperationException();
    }

    public void printMatrix() {
            for(int i=0; i<rows; i++) {
                try {
                    for(int j=i; j<columns-1; j++)
                        System.out.print(elementAt(i, j) + " , ");
                    System.out.println(elementAt(i, columns-1));
                }
                catch(IllegalArgumentException e) {}
            }
    }
}
```

За самостоятелна работа:

1. Да се състави следната йерархия от класове:
- Person
- Student
- Undergraduate
- Freshmann

За домашна работа №1 със срок на предаване 08.03.2009г.

Задача на 1 група: Функцията **f(n)** е дефинирана в множеството на целите положителни числа по следния начин:

$$f(n) = \begin{cases} 1, n = 1 \\ \sum_{i=2}^{n} f(n/i), n \geq 2 \end{cases}$$ С **n/i** е означено частното от целочисленото деление

a) Да се напише рекурсивен метод, който пресмята стойността на функцията;
b) Да се напише итеративен метод, който пресмята стойността на функцията.

Задача на 2 група: Функцията **f(n)** е дефинирана в множеството на целите положителни числа по следния начин:

$$f(n) = \begin{cases} 1, n \leq 2 \\ \sum_{i=2}^{n-1} f(n\%i + 1), n \geq 3 \end{cases}$$ С **n%i** е означен остатъка от делението на числата

c) Да се напише рекурсивен метод, който пресмята стойността на функцията;
d) Да се напише итеративен метод, който пресмята стойността на функцията.