Interfaces, Inheritance, lambda expressions

Objectives:

- using Java interfaces
- understanding and applying the inheritance mechanisms
- defining and using lambda expressions

Java Interfaces

The software entities called interfaces are specific to the Java programming language (they have no equivalent in C/C++).
A Java interface is defined using the keyword *interface*.
Syntax:

```
[access_specifier] interface InterfaceName{
      //interface body
}
```

The rules regarding the access specifier are the same as for the classes. An interface can be marked as *public* or *default* (in which case nothing is specified in the position of the access specifier).
Java interfaces are stored in *.java source files. If the interface is public, the filename must match the interface name (analogous to public classes).

A Java interface may contain:

1. variables marked with *public*, *static* and *final* access specifiers (static constants); omitting these visibility specifiers is allowed, the programming environment implicitly marks them this way.
1. declarations (prototypes) of methods, without implementation
   - implemented methods marked with the keyword `default`
   - implemented methods marked with the keyword `static`

All members of the interfaces are implicitly considered *public*. Although the initial interfaces contained only method statements, it is not necessary to use the *abstract* specifier.

Functional interfaces

The Java programming language introduces the term functional interface to describe the interfaces that contain a single method prototype.

Classes inheritance

In the Java programming language only simple inheritance is allowed, a class can be derived from a single base class. The keyword used to specify inheritance is *extends*.

Syntax:

```
class DerivedClassName extends BaseClassName{
      //derived class implementation
}
```

To compensate the shortcomings of simple inheritance, Java provides a pseudo-mechanism of multiple inheritance through interfaces: a class can implement any number of interfaces.

Syntax:

```
class DerivedClassName extends BaseClassName implements Int1, Int2, IntN{
```

```
        //derived class implementation
}
```

A derived class inherits from the base class all members marked with visibility specifiers *public* and *protected*. *Private* members are not propagated through inheritance.

Upcasting and downcasting

The *casting* mechanism allows to transform an object from one data type to another compatible one. Class inheritance hierarchies generate compatible types.

```
        class MTB extends Bike{
        //...
}
```

The instantiation mechanism of the derived or base class is already known:

```
Bile b0 = new Bike);
MTB b1 = new MTB();
```

The *upcasting* mechanism refers to the generalization of a derived class object, its type being transformed into that of the base class. This mechanism is implicit and does not require any casting.

```
// b1 is transformed into b0 and loses access to b1 specific methods
      b0 = b1; //by assignment
```

or

```
/* the anonymous instance of the derived class is converted to the base class type
and the value is assigned to object b2 */
      Bike b2 = new MTB();//by initialization
```

Downcasting mecanism refers to the granting of new facilities to an object from a base class by transforming it into an object from a derived class. This mechanism only goes using *the casting ()* operator and is restricted to classes that are in the same inheritance hierarchy.
Only objects that have been converted by upcasting can be converted again by downcasting.

```
MTB b3 = (MTB)b0;
```

Overriding the methods of a base class

Java allows redefining the implementation of methods inherited from a base class if the signature is preserved (returned type, name and list of parameters). This mechanism is called *redefinition* or *overridding*.
There is a restriction on the visibility specifier of the overwritten method, meaning: *a method cannot be overridden to restrain its visibility*.
It is recommended that the redefined methods are preceded by the *@Override* annotation.

Accessing the base class members

It is possible to access super-class members from a derived class. This can be done using the keyword *super*, the "." operator and the name of the method to be called.

A base class constructor can be called explicitly from a derived class constructor. This is possible using the same *super* keyword, and the call must be made **before any other instruction in** the derived class constructor.

Interfaces inheritance

The Java programming language allows inheriting the interfaces. Unlike classes, an interface can inherit any number of base interfaces. The keyword that specifies the inheritance of interfaces is identical to that of classes, namely *extends*.

Syntax:

```
interface DerivedInterfaceName extends Int1, Int2 [, ...]{
    //declarations
}
```

Lambda expressions

They appeared as an alternative to using the functional interfaces, to eliminate the use of anonymous classes and vertical code writing. They can be considered as being anonymous methods.

Syntax:

```
(parameters_list) -> body
```

- list of parameters - parameters ([type] and name, separated by ,)
- body - a single functional block, delimited or not by {}
- returned values can be specified

Exemple

Ex. 1 – Java interfaces

```
interface Int1{
    void metoda1(); //prototip de metoda
}

interface Int2{
    void metoda2(); //prototip de metoda
}

class Class1 implements Int1, Int2{
    //trebuie implementate metodele primite din ambele interfete
    public void metoda1(){
        System.out.println("implementarea metodei 1");
    }
    public void metoda2(){
        System.out.println("implementarea metodei 2");
    }
}

class Test{
    public static void main(String a[]){
        Class1 ob1 = new Class1();
        ob1.metoda1();
        ob1.metoda2();
    }
}
```

Ex. 2 – Java interfaces, member types

```
interface Interface_06{
   static final int max = 200;
   void metoda();
   default void metodaDefault(){
        System.out.println("mesaj din metoda default");
   }
   static void metodaStatica(){
```

```
            System.out.println("mesaj din metoda statica");
      }
}
class InterfaceUser implements Interface_06{
      public void metoda(){
            System.out.println("implementare metoda");
      }
}
class Test {
      public static void main(String args[]) {
            InterfaceUser ob1 = new InterfaceUser();
            ob1.metoda();
            ob1.metodaDefault();

            Interface_06 int1 = new InterfaceUser();//creare referinta de lucru
            Interface_06.metodaStatica();
            int1.metodaDefault();
            //int1.metodaStatica();
      }//main
}//class
```

Ex. 3 – Functional interfaces and lambda expressions

```
interface MathOperation_i{
      int operation_i(int a, int b);
}
interface MathOperation_d {
      double operation_d(double a, double b);
}
interface GreetingService {
      void sayMessage(String message);
}
public class Java11Tester {
      private int operate_i(int a, int b, MathOperation_i mathOperation) {
            return mathOperation.operation_i(a, b);
      }
      private double operate_d(double a, double b, MathOperation_d mathOperation){
            return mathOperation.operation_d(a, b);
      }
//lambdas without parenthesis
      static GreetingService greetService1 = message -> System.out.println("Hello
      " + message);
//lambdas with parenthesis
      static GreetingService greetService2 = (message) ->
      System.out.println("Hello " + message);

      public static void main(String args[ ]) {
            Java11Tester tester = new Java11Tester();
            //with type declaration
            MathOperation_i addition = (int a, int b) -> a + b;
            //without type declaration
            MathOperation_i subtraction = (a, b) -> a - b;
            //with return statement along with curly braces
            MathOperation_i multiplication = (int a, int b) -> { return a * b; };
            //without return statement and without curly braces and var type
            MathOperation_d division = (var a, var b) -> a / b;//Java 11 var types
```

```
            System.out.println("10 + 7 = " + tester.operate_i(10, 7, addition));
            System.out.println("10 - 7 = " + tester.operate_i(10, 7,
            subtraction));
            System.out.println("10 x 7 = " + tester.operate_i(10,7,
            multiplication));
            System.out.println("10 / 7 = " + tester.operate_d(10, 7, division));
            greetService1.sayMessage("World");
            greetService2.sayMessage("Java");
      }//main
}
```

Ex. 4 Inheritance, methods overriding

```
//clasa de baza
class Bicicleta{
      public void metoda(){
            System.out.println("mesaj din clasa de baza");
      }
}

//clasa derivata
class BicicletaDeTeren extends Bicicleta{
      //suprascrierea metodei din clasa de baza
      @Override
      public void metoda(){
            super.metoda(); //apelul metodei originale
            System.out.println("mesaj din clasa de derivata");
      }
}

class Test{
      public static void main(String[] args){
            Bicicleta b0 = new Bicicleta();
            BicicletaDeTeren b1 = new BicicletaDeTeren();

            b0.metoda();
            b1.metoda();
      }
}
```

Ex. 5. Inheritance, base class constructor call

```
import java.util.Random;

class Car {
        int top_speed;
        int fuel_in_gallons;

        Car(int inSpeed, int inGallons){
                top_speed = inSpeed;
                fuel_in_gallons = inGallons;
        }

        void howMuchFuel(){
```

```
                System.out.println("There is " + fuel_in_gallons +
                                        " left.");
        }
}

class Honda extends Car {
        String model;
        Honda(String theModel, int inSpeed, int inFuel){
                super(inSpeed, inFuel);   //apel constructor din clasa de baza
                model = new String(theModel);
        }
      // redefinire (suprascriere) metoda howMuchFuel()
        @Override
        void howMuchFuel(){
                System.out.println("My Honda " + model +
                                " has " + fuel_in_gallons +
                                " left.");
        }
}

class Dodge extends Car {
        String model;
        Dodge(String theModel, int inSpeed, int inFuel){
                super(inSpeed, inFuel);
                model = new String(theModel);
        }

        @Override
        void howMuchFuel(){
                System.out.println("My Dodge " + model +
                                " has " + fuel_in_gallons +
                                " left.");
        }
}

class Test {
        static Random dice = new Random();

        static Car get_a_car(){
                if (dice.nextInt() % 2 == 0)
                        return new Dodge("Caravan",90,20);
                else
                        return new Honda("Civic",70,10);
        }

        public static void main(String args[]){
                Car theCar;
                for (int i=0; i < 5; i++)
                {
                        theCar = get_a_car();
                        theCar.howMuchFuel();
                }
        }
}
```

Ex.6. Inheritance, self-reference, redefinition of the toString() method

```
class Point{
      public int x,y;

      public Point()
      {
            this(0,0);
      }

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

      public String toString()
      {
            return("("+x+","+y+")");
      }
}

class Cerc extends Point{
      public Point centru;
      public int raza;

      Cerc(int x,int y,int r)
      {
            centru=new Point (x,y);
            raza=r;
            System.out .println ("Cerc de centru "+centru+" si raza= "+r);
      }


      Cerc()
      {
            this(0,0,10);
      }

      public String toString()
      {
            return("Cerc de centru "+centru+" si raza= "+raza);
      }
}

class Linie extends Point {
      public Point p1,p2;

      Linie()
      {
            this(0,0,0,0);
      }

      Linie(int x1,int y1,int x2,int y2)
      {
            this(new Point (x1,y1),new Point (x2,y2));
      }
```

```
      Linie(Point p1,Point p2)
      {
            this.p1=p1;this.p2 =p2;
            System.out .println ("Linie de la "+p1+" la "+p2);
      }

      public String toString()
      {
            return("Linie: "+p1+"-"+p2);
      }
}

class Triunghi extends Linie {
      public Linie l1,l2,l3;

      Triunghi()
      {
      this(0,0,0,0,0,0);
      }

      Triunghi(int xl1,int yl1,int xl2,int yl2,int x22,int y22)
      {
      this(new Linie(xl1,yl1,xl2,yl2),new Linie(xl2,yl2,x22,y22),new
Linie(x22,y22,xl1,yl1));
      System.out.println("Linie de la ("+p1.x+","+p1.y+") la
("+p2.x+","+p2.y+")");
      }

      Triunghi(Point p1,Point p2,Point p3)
      {
            this(new Linie(p1,p2),new Linie(p2,p3),new Linie(p3,p1));
      }

      Triunghi(Linie l1,Linie l2,Linie l3)
      {
      this.l1=l1;this.l2=l2;this.l3=l3;
      System.out.println("Triunghi");
      }

      public String toString()
      {
            return("Triunghi cu  :\n\t"+l1+"\n\t"+l2+"\n\t"+l3);
      }
}


class Test{

      public static void main (String[] args){
            Point p1,p2,p3;
            Cerc c1,c2;
            Linie l1,l2,l3;
            Triunghi t1,t2,t3,t4;

            p1=new Point (1,2);
            p2=new Point (1,12);
            p3=new Point (1,20);
```

```
            c1=new Cerc ();
            c2=new Cerc (1,2,3);

            l1=new Linie();
            l2=new Linie(1,2,1,15);
            l3=new Linie(p1,p2);

            t1=new Triunghi();
            t2=new Triunghi(1,1,1,10,5,7);
            t3=new Triunghi(p1,p2,p3);
            t4=new Triunghi(l1,l2,l3);


            System.out.println ("\nFigura contine urmatoarele elemente:\n");
            System.out.println(c1);
            System.out.println(c2);

            System.out.println(l1);
            System.out.println(l2);
            System.out.println(l3);

            System.out.println(t1);
            System.out.println(t2);
            System.out.println(t3);
            System.out.println(t4);
      }
}
```

Ex. 7. – Inheritance, upcasting, downcasting

```
class Camion{
protected String sigla;
protected String nume;
      public Camion(){
            this.nume = "fara nume";
            this.sigla = "camion";
      }
      public Camion(String nume){
            this.nume = nume;
            sigla = new String("camion");
      }

      public String returneazaInfo(){
            return "ma numesc "+nume+" si sunt un "+sigla;
      }
      public void metodaSpecificaCamion(){
            System.out.println("CAMION!!!");
      }
}

class Volvo extends Camion{
//variabilele se mostenesc, nu mai trebuie re-declarate
//String sigla;
//String nume;
      public Volvo(String nume){
            this.nume = nume;
```

```
            sigla = new String("Volvo");
      }
//suprascrierea metodei originale

      @Override
      public String returneazaInfo(){
            return "numele meu este "+nume+" si sunt un "+sigla;
      }
      public void metodaSpecificaVolvo(){
            System.out.println("VOLVO!!!");
      }
}

class Test{
      public static void main(String a[]){
            Camion c1 = new Camion("c1");
            System.out.println(c1.returneazaInfo());
            c1.metodaSpecificaCamion();
            Volvo v1 = new Volvo("v1");
            System.out.println(v1.returneazaInfo());
            v1.metodaSpecificaVolvo();
            Volvo v2 = new Volvo("v2");
            Camion c2 = v2; //upcasting prin initializare
            System.out.println(c2.returneazaInfo());
            c2.metodaSpecificaCamion();
            //c2 nu are acces la metodaSpecificaVolvo
            //c2.metodaSpecificaVolvo();
            Volvo v3=new Volvo("XC90");
            System.out.println(v3.returneazaInfo());
              //downcasting
            if(c2 instanceof Volvo) {
                  v3=(Volvo)(c2);
                  System.out.println("Dupa downcasting " +v3.returneazaInfo());}
            v3.metodaSpecificaVolvo();
            v3.metodaSpecificaCamion();//e mostenita
            Camion c3 = new Camion();
            Volvo v4 = (Volvo)c3; //ClassCastException
            v4.metodaSpecificaVolvo();
            v4.metodaSpecificaCamion();
      }
}
```

Individual work

Complexity *

1. Consider a Java interface that contains the prototypes of the methods of addition, subtraction, multiplication, division, square root and raising a number to a certain power. All methods will have two double type parameters and specify the returned double type.
Implement the interface so that operations are defined within a class. Instantiate the class and check the implemented operations.

1'. Instead of an interface that contains all the aforementioned methods, implement 4 functional interfaces, one for each method.
The interfaces will be implemented within a single class. Instantiate the class and check the implemented operations.

1''. Use the structure used at 1' and implement lambda expressions to define the arithmetic operations. Check the functionality.

2. Define an interface called *GeometricForm* that contains methods which return the area and perimeter of the geometric form. Implement the interface for: squares, rectangles, circles, equilateral triangles and isosceles triangles. Read from the keyboard N distinct geometric forms specified by their type and specific parameters (for example for a circle, you would need to read its radius). Compute the total area and perimeter of all the geometric forms, taking into account the fact that they do not overlap.

Complexity **

3. Develop a class hierarchy of shapes and write a program that computes the amount of paint needed to paint different objects. The hierarchy will consist of a parent class Shape with three derived classes - Sphere, Rectangle, and Cylinder. For the purposes of this exercise, the only attribute a shape will have is a name and the method of interest will be one that computes the area of the shape (surface area in the case of three-dimensional shapes). Do the following.
A. Write an abstract class Shape with the following members:
- an instance variable shapeName of String type
- an abstract method area()
- a toString() method that returns the name of the shape
B. The file Sphere.java contains the Sphere class which is a descendant of Shape. A sphere has a radius as a private integer variable and implements the body of the inherited abstract method area().
C. Define similar classes for a rectangle and a cylinder. The classes Rectangle and Cylinder are also derived from the Shape class. A rectangle is defined by its length and width. A cylinder is defined by a radius and height.

Note: Each of the derived classes override the toString method and define specific mutator and accessor methods.

D. The file Paint.java contains an interface that has a float constant paintPerSurfaceUnit.
E. The file PaintThings.java implements the Paint interface and contains a program that computes the amount of paint needed to paint various shapes.
Instantiate the three shape objects: deck <- Rectangle, bigBall <- Sphere and tank <- Cylinder. Make the appropriate method calls to assign each object's specific attributes with data read from the keyboard.
Compute the amount of paint needed for covering each created shape.

4. Consider the *Fraction* class that has two protected attributes *a* and *b* for the counter and denominator, two *set ()* and *get ()* methods for each of the class attributes. Define an explicit constructor without parameters that initiates *a* with 0 and *b* with 1, and an explicit constructor with two parameters that can be called if it is checked whether a fraction can be defined (*b*! = 0). Define a method *simplify ()* that simplifies and returns a *Fraction* object by calling the *int greatestCommonDivider (int, int*) method (based on divisions). Define a method for adding two *Fraction* objects, which returns a *Fraction* object. Define a *ExtendedFraction* class derived from *Fraction*, which will have a constructor with parameters (which calls the constructor from the base class) and which will redefine the method *simplify()* using an *int greatestCommonDivider (int, int)* algorithm based on subtractions. Add a method for subtracting two fractions. Instantiate two *Fraction* objects without parameters. Set the attributes of the data objects read from the keyboard. Display the original attributes of the objects and the new defined attributes. Simplify, add and display results. Instantiate two *ExtendedFraction* objects with data read from the keyboard. Simplify, add and subtract objects and display results. Make an upcast from *ExtendedFraction* to *Fraction* and try to subtract the items. All operations will be called from the *main()* method.

5. Define a String Array. Using lambda expressions sort it by the following criteria: length (small->large), inverse length (large->small), alphabetical order, the strings which start with the letter M are first, then come the rest.

Complexity ***

6. Define an interface called *ChessPiece* which defines the prototype for the method move(). Create the specific classes for each of the chess pieces and implement the move() method according to the rules of movement on the chess board. The method takes as input the chess piece's current location and the direction of the move given by the geographical coordinates (N, S, E, V, NE, NV, SE, SV), and returns the final position of the piece. Pay attention to the pieces which can move a different number of cells!!

7. Define a class called *Vehicle* which has as attributes the maximum number of passengers, its color and the maximum speed. Extend the *Vehicle* class within the *MotorizedVehicle* class which also includes the geo-location attributes (GPS coordonates – create a class for this type of object GeoLoc) and the moving speed of the vehicle. *MotorizedVehicle* can move from point A to point B (the points are specified using a Point object which has two *GeoLoc* attributes) by using the move(Point B) method and which returns the total duration of the trip.
Create a new class called *Airplane* which extends the *MotorizedVehicle* class and which add the attribute altitude to the move(Point B) method. This method will return the time needed to take the trip by taking into account that the plane will travel on arc of a circle specified through points A and B and the maximum altitude reached by the plane (the maximum altitude is reached half-way between A and B).