

# Introduction to Programming

## Exercises WS 03-04

### Solutions

#### 1. The main function.

##### a) What is the main function?

Every Java application contains a class, with a method (function) called `main()` being defined. Classes may be called arbitrarily, but the `main()` method always has a fixed form (see below). When the Java Virtual Machine (JVM) is activated (by the `java classname` command), it must be supplied with the name of a class (`classname`). To get it, the JVM looks for a default method called `main` in the class with this name and starts running a program from here. In this sense, the `main` method (function) is often said to be the entry point to the program. The `main` method is necessary to run a program; if it is missing an error message is displayed, and if it is not having the particular required form it will not be recognized by the JVM as the method where the program execution starts.

##### b) Write the signature of the main method.

```
public static void main (String [ ] args)
```

The signature consists of a method name and of a list of parameter types. In some Java textbooks the signature of the main method is defined as

```
void main (String [ ] args)
```

##### c) Explain the meaning of all the words in the statement containing the main function.

The word `public` is an access modifier. Since the `main` method must be accessible by the JVM, i.e. by an external code, it should be made visible to the outside classes, which is ensured by the keyword `public` preceding the method declaration.

The keyword `static` signifies that the `main` method can be invoked without the need to generate instances of the class, i.e. without instantiating objects. Why is it necessary to have the keyword `static` preceding `main`? Because the `main` method is called by the JVM before any objects are created.

The keyword `void` instructs the compiler that the `main` method does not return any values – this method is intended for other purposes, e.g. to launch a program.

The `main` method, like any other method, must have parameters – similarly to any function that must have variables. Here, the parameter is in the form ensuring that information can be transmitted to the program when the latter starts up. The square brackets indicate that the parameter is going to be an array, i.e. a number of elements having the same name and

indexed by numbers. The declaration `String [ ] args` designates a set of arguments `args` being an array of instances of the `String` class. An array of strings is reserved for input arguments - in this array strings of characters are stored. The idea is that an array of strings `args` would receive all the command line input parameters and pass them to the JVM, when the program is run. One must notice that all command line arguments are passed as `String` values, never as numbers.

Curly brackets “{” and “}” designate respectively the start and finish of the main method. In general, one might similarize these symbols to “do” and “od”, “if” and “fi” in Maple procedures.

## 2. Compiler and interpreter

### a) What is the difference between a compiler and an interpreter?

**A compiler.** The process of programming implies creating a sequence of commands to the compiler, written in a particular programming language (a high-level language). However, a CPU, which basically consists of a great number of binary switches, can directly execute only machine language commands, the latter being designed for a specific CPU type. Thus, a systematic process should exist allowing to map a program in a high-level language into a machine language program, both implementing the same function. Such map is called compilation (from the Latin *compilatio* – theft, abduction) and the program that performs the map is called a compiler.

The compiler fulfils two functions:

- 1) checking the input (source) code for syntax errors. If there are any errors in the source code, the latter should be corrected and resubmitted for compilation.
- 2) translating the high-level language instructions into the machine code, i.e. a native form that can be run on a specific type of computer.

**An interpreter** is a special program in the machine-code language that emulates direct execution of the input high-level language program.

**The difference:** the compiler maps the whole input program into the machine language program, so that the complete source code is translated into the machine code, whereas the interpreter translates each instruction and executes it in “real time” (run-time).

### b) How is the compiler-interpreter dichotomy implemented in Java?

Compilers for most programming languages translate the source code into the native machine language, which is specific for a given CPU. In Java, the source code is mapped into the JVM, irrespective of the platform architecture. The result of the compilation is not the executable code, but some standard code called the Java bytecode, which can be run on different platforms by the Java interpreter, the latter being platform-dependent.

The Java compiler can be called by typing in the terminal window:

```
javac filename.java
```

The Java interpreter executing the compiled program is called by typing in the following command:

java filename,  
where filename.java is the file that has been created by an editor.

### c) What are the advantages of Java's hybrid compiler + interpreter approach?

The principal advantage of the Java hybrid compiler+interpreter approach is platform independence. The interpreter is contained within the JVM; it translates the bytecode in real time and runs the application on a particular machine. Thus, theoretically, only the interpreter should be ported. A program developed e.g. on a Windows desktop can be compiled on a UNIX, Linux, Mac, etc., since the standard bytecode can be exchanged between these machines. Having the shared bytecode, respective interpreters – platform dependent – can execute programs on a particular platform.

The Java hybrid approach also implies a certain disadvantage, namely the Java code is not optimized for a specific platform. This fact may lead to a slowness of the Java code as compared to the one being optimized for a given platform.

### Why does the idea of bytecodes look so appealing to Java developers?

Java has been developed predominantly for the Web access by using applets. The latter can be run by the user with the help of interpretation and not by full compilation of programs. Thus, even very simple Internet-based devices, e.g. without hard disk storage, can be operated with the help of Java. The necessary condition for it is enabling of the JVM in Web browsers.

## 3. The first program in Java

### a) Write a program that produces the following output:

**Hello World!**  
**Bonjour Monde!**  
**Vale Mundum!**

The output consists of three lines of text, so basically it needs three `System.out.println( )` statements:

```
public class Greetings
{
    public static void main (String [ ] args)
    {
        System.out.println ("Hello World!");
        System.out.println ("Bonjour Monde!");
        System.out.println ("Vale Mundum!");
    }
}
```

One may improve the code using the `\n` escape character denoting “new line”:

```

public class Greetings
{
    public static void main (String [ ] args)
    {
        System.out.println ("Hello World!\n Bonjour Monde!\n Vale Mundum!");
    }
}

```

A more advanced way to write the program producing the same output without repeated println calls would be

```

public class Greetings
{
    public static void main (String [ ] args)
    {
        sayHello ("Hello World!");
        sayHello ("Bonjour Monde!");
        sayHello ("Vale Mundum!");
    }
    public static void sayHello (String string)
    {
        System.out.println(string);
    }
}

```

### **b) Could you produce the same output in Arabic, Chinese, Russian?**

Java is an English-based language; in fact the source code looks like stylized English. It means that one can only translate the string, not the code itself. In the first releases of Java Arabic, Chinese, Russian, etc. were not supported due to the specificity of alphabet. However, many languages using non-Latin based characters are supported in the new Java releases with the help of the Unicode character settings (see <http://www.unicode.org>). The Unicode is the universal system of characters for all today's world languages, with the capacity of  $2^{16} = 65536$  symbols (now approximately one half of Unicode places are occupied). With the appropriate Unicode characters, the bytecode generated by compiling the source of the Greetings type can be correctly interpreted. We only need to add lines with greetings in respective languages.

### **c) What would happen if you changed the name of the source file, e.g. HelloEarth.java instead of Test.java?**

The name of the file (filename.java) should coincide with the name of the class described in the file. If one arbitrarily changes the filename, the compiler will throw an error, (usually NoClassDefFound Error)

## **4. Producing an output**

a) What is the output of the following program?

// this is the Hello Simon program in Java

```
class HelloSimon {
    public static void main (String [ ] args) {
        String name = "Simon";

        /* Now let us say hello */
        System.out.println ("Hello" + name);

        /* or another variant */
        System.out.println ("Hello + name");
    }
}
```

Hello Simon  
Hello + name

b) What would the following program print? Explain the output

// This is the Guten Tag program in Java

```
class GutenTag {
    public static void main (string args [ ] ) {
        String name = "Professor Zenger";
        /* Now let us say Guten tag */
        System.out.println("Guten Tag + name");
    }
}
```

Guten Tag + name

The second quotation mark should be after the word **Tag**, not after **name**. The word **name** is not recognized as a variable inside a string. Moreover, unless the word **string** in the **main** method is corrected to **String**, the compiler will throw an error (**cannot resolve symbol**). This happens because Java is a case-sensitive language, so in this case the Java compiler cannot recognize the string in the **main** method.

In the present case, the error appeared due to careless use of MS Word, which automatically makes spelling corrections, e.g. case changes, unless this feature is turned off. One may notice the same kind of error in the commentary that follows.

**5. Write a program that prints all the integers between 0 and 17.**

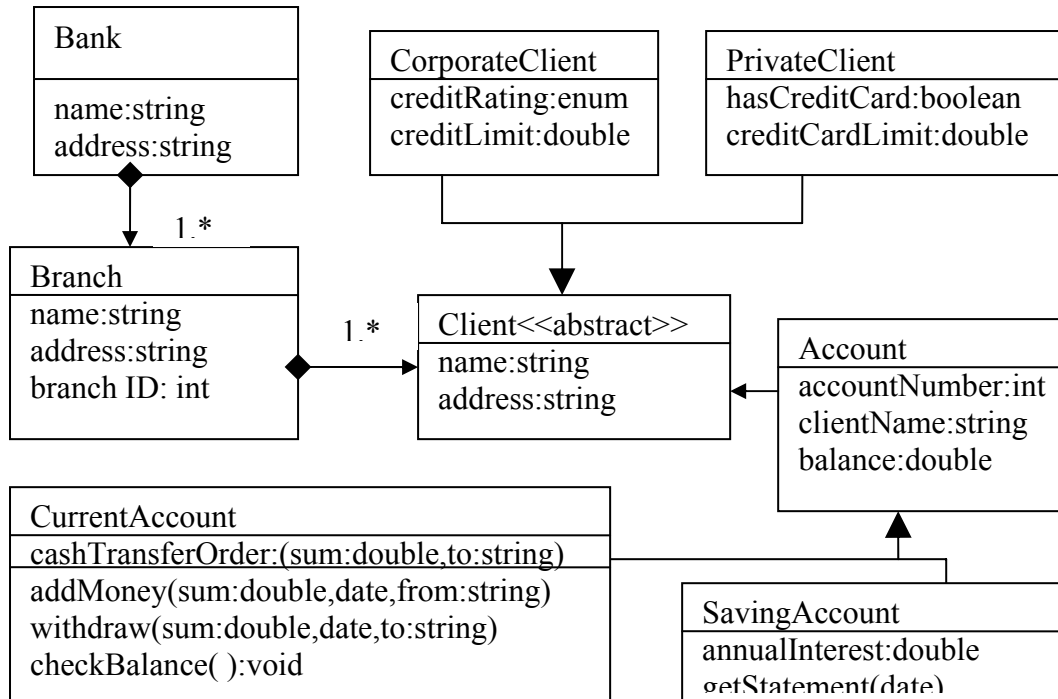
```
// a while-loop version
public class Seventeen
{
    public static void main (String [ ] args)
    {
        int i;
        i = 0;
        while (i < 18)
        {
            System.out.println ( i );
            i = i +1;
        }
    }
}

// a for-loop version

public class Seventeen
{
    public static void main (String [ ] args)
    {
        for (int i = 0; i <= 17; i++)
        {
            System.out.println(i + “.”);
        }
    }
}
```

**6. A modeling task: the banking project.**

**a) Construct an object-oriented model of a bank that manages its clients’ accounts.**



**b) What attributes does the Account class have?**

For example:

```

private int accountNumber;
private String clientName;
protected double balance;

```

Some other, like e.g. the PIN code and the opening date may be incorporated in the full set of necessary attributes.

**c) What methods should one define in order to access the class attributes and what limitations should one consider while invoking these methods?**

The accessor methods such as the “get” method, in order to access the attributes of the Account class, for instance:

```

public int getAccountNumber( )
{
    return accountNumber;
}

```

or

```

public String getClientName( )
{
    return clientName;
}

```

These methods are declared **public**, which ensures that the object can access them. In the banking system; the attributes should be only read and not set up from outside. They can be set by calling the constructor method of the **Account** class with appropriate parameters (see the example below). In general, one has to guarantee that only the client object should have an access to sensitive information, e.g. to invoke the **withdraw** method

An example of the source code for the **Account** class may look as follows:

```
public class Account
{
    int accountNumber; // account identifier
    int amount;        // available amount of money
    int pincode;       // account pincode

    //Constructor
    public Account(int accountNumber)
    {
        this.accountNumber = accountNumber;
        this.pincode = (new java.util.Random()).nextInt(20); // generate random
pincode
    }

    // Withdraw
    public void withdraw(int amount)
    {
        if (amount <= this.amount) // Constraint: test the availability of the desirable
// amount
            this.amount = this.amount-amount;
        else
            System.out.println("error, the withdraw amount must be lower as the
available amount");
    }

    // Deposit
    public void deposit(int amount)
    {
        this.amount = this.amount+amount;
    }

    // Test the correctness of the pincode
    public boolean checkPincode(int code)
    {
        if(code == this.pincode)
            return true;
        else
            return false;
    }

    // Returns the account identifier
```

```

    public int getAccountNumber()
    {
        return accountNumber;
    }

    // Returns the pincode

    public int getPincode()
    {
        return pincode;
    }

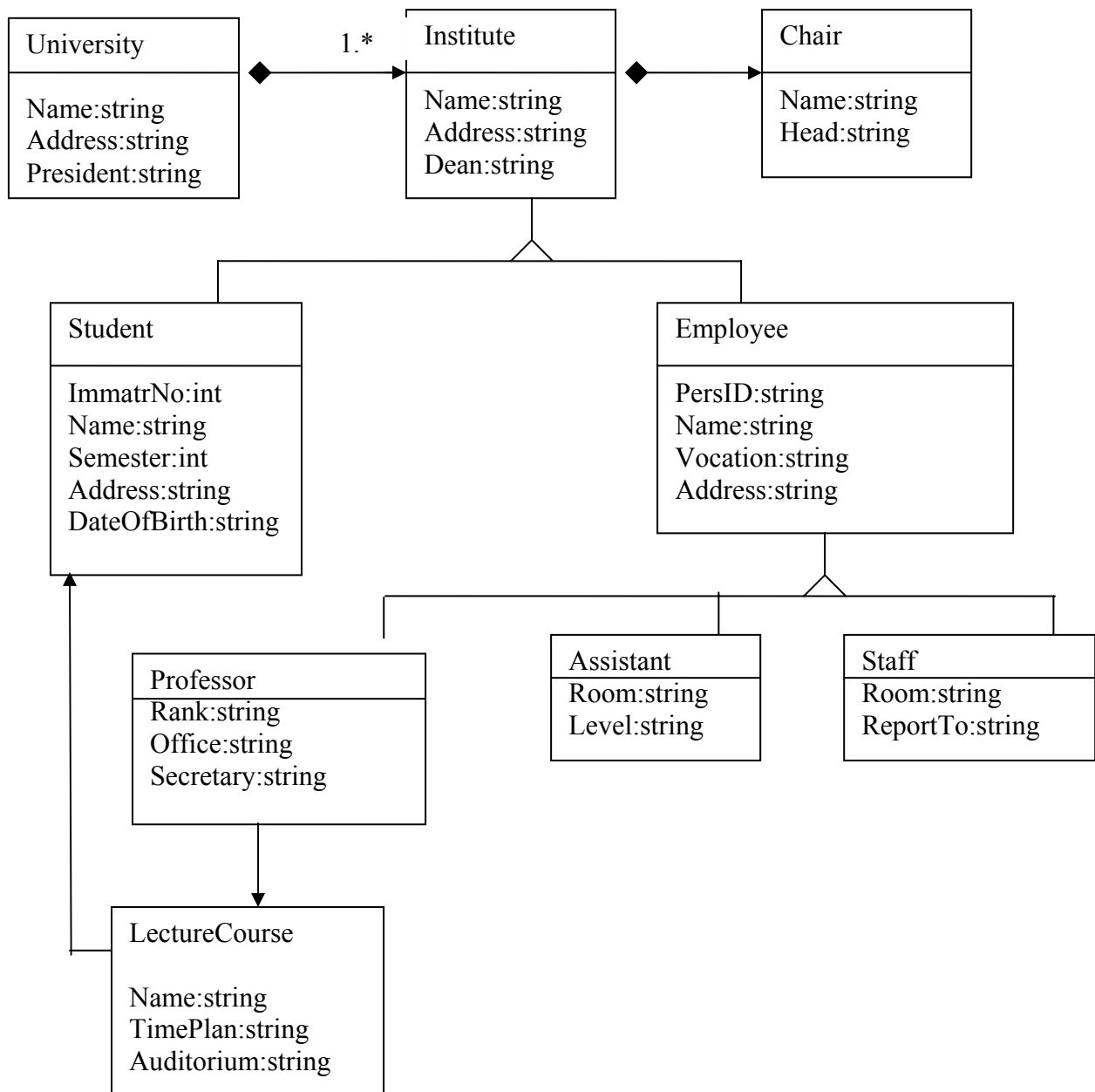
    // The withdrawal output

    public void printStatement()
    {
        System.out.println("*****");
        System.out.println("The account No "+String.valueOf(accountNumber)+" :");
        System.out.println("    available amount =" +String.valueOf(amount)+".");
    }
}

```

**7. Construct a model of a University. You can take the TU München as an example.**

The class diagram:



You may see the respective code examples in the script “Introduction to Programming” by D.Chibisov, S.Pankratov, C.Zenger

## 8. The Fibonacci recursion

**a) Write a program that would print the first 20 Fibonacci numbers. (A reminder: the Fibonacci numbers are defined by the following recurrent relation:**

$$f_n = f_{n-1} + f_{n+1}, f_0 = 0, f_1 = 1 )$$

The code may look, for example, as follows (the while-loop version):

```

public class Fibonacci
{
    public static void main(String [ ] args) {
        int i;
        int fibold;
        int fibnew;
        int temp;

        i = 0;
        fibold = 1;
        System.out.println(fibold);
        fibnew = 1;
        System.out.println(fibnew);

        while (i < 19)
        {
            temp = fibold + fibnew;
            fibold = fibnew;
            fibnew = temp;
            System.out.println(fibnew);
            i = i + 1;
        }
    }
}

```

A for-loop version:

```

public class Fibonacci
{
    public static void main (String [] args)
    {int fibold = 0, fibnew = 1, temp;
        for(int i = 2; i<=20; i++)
        {
            temp = fibold + fibnew;
            fibold = fibnew;
            fibnew = temp;
        }
        System.out.println(fibnew);
    }
}

```

One can also notice the difference between iteration and recursion

Iteration:

```
fibold = 0;
fibnew = 1;
for (i=2; i<=n; i++) {
    temp = fibold + fibnew;
    fibold = fibnew;
    fibnew = temp;
}
```

Recursion:

```
int F(int n)
{if (n==0) return 0;
  else if (n==1) return 1;
  else return F(n-1)+F(n-2);
}
```

One can see that iterative solution is faster and requires less memory, because the previous Fibonacci numbers are not necessarily stored in the iterative procedure. Contrariwise, since  $F(n)/F(n-1) \sim 1.618$  (the Golden Section), one has to call the function  $F$  over  $1.6^n$  times. If  $n \gg 1$ , recursion becomes meaningless.

#### b) Compare the Java code with the Maple code.

The Maple code may be, for example, as follows:

```
> restart;
> F := proc(n) F(n-1)+F(n-2) end proc;
      F(0) := 0;
      F(1) := 1;

      F := proc(n) F(n-1) + F(n-2) end proc
      F(0) := 0
      F(1) := 1

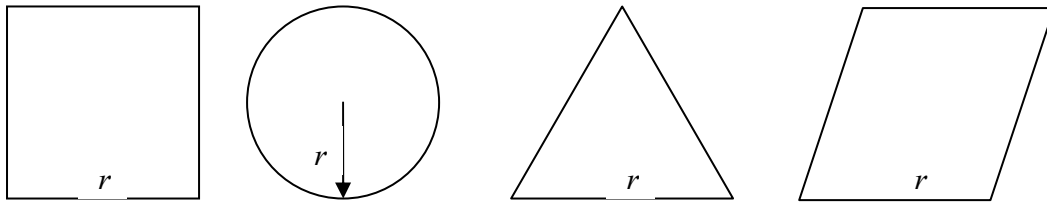
> F(20) ;
      6765
```

Moreover, there is a built-in function “fibonacci” in Maple that computes the Fibonacci numbers

```
> seq(fibonacci(i), i=0..19) ;
```

So, the Maple code seems to be easier and more straightforward.

9. An engineering company would like to compare the areas of different shapes, namely of a square, circle, equilateral triangle and rhombus with  $45^\circ$  angle. The characteristic measurement for all the shapes is  $r$ :



Write a program that would print the area values in  $m^2$  for  $r = 20$  m

```
public class Area {
    public static void main (String [ ] args) {
        int r=20;
        double areaSquare = r*r;
        double areaCircle = Math.PI*r*r;
        double areaTriangle = Math.sqrt(3)*(r*r/4);
        double areaRhombus = Math.sqrt(2)*(r*r/2);
        System.out.println("The area of the square is "+areaSquare+" sq.m");
        System.out.println("The area of the circle is "+areaCircle+" sq.m");
        System.out.println("The area of the triangle is "+areaTriangle+" sq.m");
        System.out.println("The area of the rhombus is "+areaRhombus+"sq.m");
    }
}
```

10. There are 454 grams in a pound.

a) Write programs that would convert pounds to kilograms and kilograms to pounds.

**Pounds to kilograms:**

```
public class Lbstokg
{
    public static void main (String [ ] args)
    {
        double lbs = Double.valueOf(args[0]) .doubleValue();
        double kg = lbs*0.454;
        System.out.println(kg +" kg");
    }
}
```

### **Kilograms to pounds:**

```
public class Kgtolbs
```

```
public class Kgtolbs
{
    public static void main (String [ ] args)
    {
        double kg = Double.valueOf(args[0]).doubleValue();
        double lbs = kg*1/0.454;
        System.out.println(lbs +" lbs");
    }
}
```

### **b) Can you make one program instead of two?**

```
public class Conversion
{
    public static void main (String [ ] args)
    {
        if (args[1].equals("kg"))
        {
            double kg = Double.valueOf(args[0]).doubleValue( );
            double lbs = kg*1/0.454;
            System.out.println(lbs +" lbs");
        }
        else
        {
            double lbs = Double.valueOf(args[0]) .doubleValue();
            double kg = lbs*0.454;
            System.out.println(kg +" kg");
        }
    }
}
```

### **c) How can you reuse this program?**

Basically, the same code may be applied to a very general task of unit conversion, e.g. miles to km, inches to cm, angle degrees to radians, between different temperature and energy scales, between various currencies, etc. This is important not only for programming the transition from one measuring system to another, e.g. from SI to Gaussian, but also for considering scaling laws and non-trivial physical relationships (such as  $E=mc^2$ ).

### **11. Write the program that would compute the equivalent resistance:**

The program should read out a series of resistances from the command line and produce the equivalent resistance as an output.

**a) For  $n$  resistances connected serially**

```
public class SerialR
{
    public static void main (String [ ] args)
    {
        double equivalent = 0;
        double resist;
        for (int i = 0; i < args.length; i++){
            resist = Double.valueOf(args[i]).doubleValue( );
            equivalent = equivalent + resist;
        }
        System.out.println("The equivalent resistance is "+equivalent);
    }
}
```

**b) For  $n$  resistances connected in parallel**

```
public class ParallelR
{
    public static void main (String [ ] args)
    {
        double equivalent = 0;
        double resist;
        for (int i = 0; i < args.length; i++){
            resist = 1/Double.valueOf(args[i]).doubleValue( );
            equivalent = equivalent + resist;
        }
        equivalent = 1/equivalent;
        System.out.println("The equivalent resistance is "+equivalent);
    }
}
```

As a primitive graphical output example, one can produce the following piece of code:

```
public class Resistance {
    public static void main(String [ ] args) {
        System.out.println("For parallel connection\n");
        System.out.println("    |-----1.3 kOhm-----|");
        System.out.println("-----|-----2.5 kOhm-----|-----");
        System.out.println("    |-----1.5 kOhm-----|");
        System.out.println( );
        System.out.println("The equivalent resistance is “ +
            (1 / (1/1.3 + 1/2.5 + 1/1.5)) + “kOhm”);
    }
}
```

**12. When writing a program, we usually start by declaring the class with a line:**

```
public class * {
```

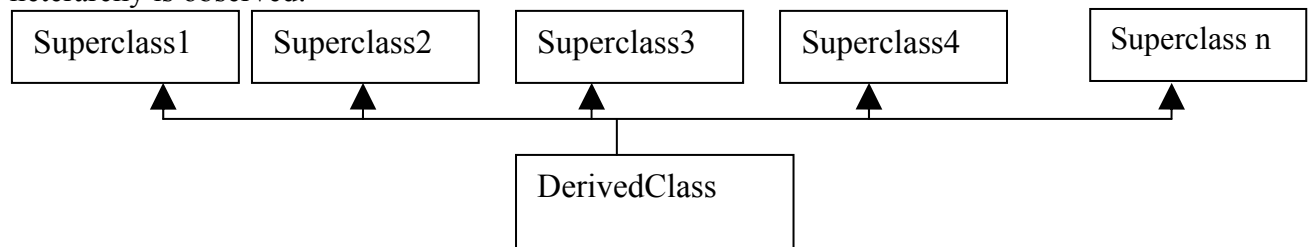
.....

**Can the class be declared private?**

No, the class cannot be declared private, because in this case it could not be accessed by the JVM, which is an external code. The compiler would throw an error, possibly “Modifier private is not allowed here”

**13. Explain the concept of multiple inheritance in OOP. Is multiple inheritance allowed in Java? In C++? In C#?**

From the formal point of view, multiple inheritance is defined as the possibility for a class to have more than one direct superclasses. In such cases, instead of class hierarchy a class heterarchy is observed:



This principle may lead to difficulties (it is often said that multiple inheritance “opens up a can of worms”). For instance, what happens when different superclasses possess different properties with the same name? From which superclass should the derived class inherit? Even more complicated situation appears when the superclasses are in their turn derivates of some class, so that the latter feature are branched and then merge again due to multiple inheritance. An example: a human is a mammal (some think a human is an ape) as well a society element, so he/she inherits features from these two superclasses (mammals and socium). The interplay of social and biological properties makes human behavior and its modeling rather complicated. Or, for instance, a student is a Human and a Personality. The simplest example is the inheritance from mother and father. Nevertheless, multiple inheritance may prove necessary to operate with mathematical objects, which often inherit properties from different superclasses (e.g. vector is a tensor and a matrix).

To keep Java as simple as possible, multiple inheritance is not allowed in this language, as well as in Smalltalk that was Java predecessor. In C++, multiple inheritance is permitted, with types of inheritance being specified: private, public and protected. The syntax for a derived class D in terms of base classes  $B_i, i=1,2..n$ :  
class D : private B<sub>1</sub>, public B<sub>2</sub>, private B<sub>3</sub>

In Java, another mechanism exists allowing one to emulate the multiple inheritance. This mechanism is based on the notion of interface.

The multiple inheritance is also not allowed in C#. The syntax:  
class D : B

**14. Explain how the Java interface works. What is the difference between an abstract class and an interface? Produce examples of “implements” relationships.**

**An interface** contains only the specification of useful methods, not the methods themselves, i.e. an interface cannot have instance variables or concrete methods. Here is an example of the definition of an interface:

```
public interface PlaneShape
{
    public abstract double getPerimeter( );
    public abstract double getArea( );
}
```

Here, the designer of the interface has no idea how the methods `getPerimeter` and `getArea` can be implemented, so the methods have no bodies. Nevertheless, the interface must guarantee that subsequent classes should implement the methods declared in the interface. In this sense, the interface acts like a treaty or a convention protocol. The classes that are to implement the interface should override all the methods declared in it, i.e. provide their own specification for the methods.

The idea of the interface concept in Java is to provide an opportunity for a polymorphic behavior throughout classes that are not located within the same hierarchy. See Ch. 10.6 in D.J. Barnes, M. Kölling. *Objects First With Java*, Pearson, 2003.

**Abstract class.** Abstraction may be understood as the separation of the logical properties of an entity (of a class or a method in the Java context) from its implementation. In Java, abstraction allows a class or a method to focus on the essential behavior and interface to the world. Details are to be filled in at a later stage. One might say that abstract methods provide a place for methods that can be mentioned in one class, but that can be implemented in various ways in other classes. For example, the concept of a mathematical function can be declared as an abstract method:

```
abstract double f (double x);
```

When the actual calculation for the method is given, one may e.g. declare:

```
double f (double z);
{
    return Math.cosine(z);
}
```

Abstract methods are contained in classes, and any class having at least one abstract method becomes abstract, since the presence of an abstract method means that such class is incomplete. An abstract class in Java cannot have any instances – it should be subclassed and the abstract methods should be filled in.

By implementing an interface, a class should provide a variant for each of the methods listed in the interface, so an interface is similar to an abstract class. However, there are two main differences:

1. Interfaces only declare methods

2. A class may implement several interfaces, but have (extend) only one parental class

A typical form of implementing the interface:

```
class Classname implements Interfacename
{
.....the body of the class
}
```

An example:

```
public class Square implements PlaneShape{
    private double side;
    double getArea( )
    {
    return side * side;
    }
}
```

So, the main difference between an abstract class and an interface is as follows. An interface can be implemented by a class, which already extends another class, whereas a class, which is already a subclass of another class, cannot additionally extend an abstract class. Moreover, an interface cannot contain non-abstract methods, while an abstract class can.

**15. Can you remove (delete) methods and fields from a subclass? Explain your thoughts.**

No, you cannot. You can only add them. If you were able to remove methods and fields from a subclass, then the latter could not be, in general, considered as an extension of its superclass. For example, in the limiting case when all the methods and fields were removed from the superclass, the subclass would be totally unrelated to the superclass.

**16. Write a program that would compute the daily pay of an employee (e.g. the program may produce an output by reading the number of hours the employee was actually working and an hourly rate).**

```
public class DailyPay
{
    public static void main (String [ ] args) {

        double hoursDay = Double.valueOf (args [0]).doubleValue( );
        double rate = Double.valueOf (args [1]).doubleValue( );
        double pay = rate * hoursDay;

        // rate = 10;
        // hoursDay = 8;
        System.out.println("The daily pay due is " + pay + " Euro ");
    }
}
```

**17. Modify the above program, taking into account the weekly working hours (WWH) limit of 40 and double pay for overwork.**

```
public class WeeklyPayPlus
{
    public static void main (String [ ] args)  {

        double hoursWeek = Double.valueOf (args [0]) .doubleValue( );
        double rate = Double.valueOf (args [1]) .doubleValue( );
        double pay;

        if (hoursWeek > 40) {
            pay = rate * 40 + 2 * rate * (hoursWeek - 40);
        }
        else {
            pay = rate * hoursWeek;
        }
        System.out.println("The weekly pay due is " + pay + " Euro ");
    }
}
```

**18. Generalize the above program, if the WWH limit and the overwork pay rate are arbitrarily changed (e.g. to 45 and ½ instead of 2). Produce some realistic examples.**

*The solution to the problem given below has been produced by Ms. Filiz Zevri*

```
/**
 * Calculation of the daily pay.
 * We assume that when creating an object of the Employee type, the following
 information is * required: the number of working hours in the week and the hourly
 rate.
 * We shall consider a normal working day to be 8 hours.
 */
```

```
public class Employee
{
    //Fields
    private double _hoursWorkedWeekly;
    private double _hourlyRate;

    // Constructor for the objects of the Employee class
    public Employee (double hourlyRate, double hoursWorkedWeekly)
    {
        // initialize instance variables
        _hourlyRate = hourlyRate;
        _hoursWorkedWeekly = hoursWorkedWeekly;
    }

    public double CalculateDailyPay_Exercise16 ( )
    {
```

```

        double dailyPay = hoursWorkedWeekly*hourlyRate/8;
        System.out.println("The daily pay is: "+dailyPay);
        return dailyPay;
    }

    public double CalculateDailyPay_Exercise17 ( )
    {
        //first check whether the employee has produced any overwork during
the week
        double overtime = _hoursWorkedWeekly – 40;
        if (overtime < 0)
        {
            System.out.println("No overtime");
            return CalculateDailyPay_Exercise16 ( );
        }
        else
        {
            double normalTimePay = 40*_hourlyRate;
            double overTimePay = overtime * 2 * _hourlyRate;
            // normally, the employee must work 8 hours a day
            double dailyPay = (normalTimePay + overTimePay) /8;
            System.out.println ("The daily pay in case of overwork is: "+dailyPay);
            return dailyPay;
        }
    }

    public double CalculateDailyPay_Exercise18 (double wwLimit, double
overworkRate)
    {
        //first check whether the employee has produced any overwork during
the week
        double overtime = _hoursWorkedWeekly – wwLimit;
        if (overtime < 0)
        {
            System.out.println ("No overtime");
            return CalculateDailyPay_Exercise16 ( )^;
        }
        else
        {
            double normalTimePay = wwLimit * _hourlyRate;
            double overTimePay = overtime * overworkRate * _hourlyRate;
            // the normal working day for an employee is 8 hours a day
            double dailyPay = (normalTimePay + overTimePay) /8;

/* The program of Exercise 18 is a generalization of the two preceding programs
(Exercise 16 and Exercise 17), since having declared the method
CalculateDailyPay_Exercise18, the method content in e.g. Exercise 17 is reduced to
a single line:
return CalculateDailyPay_Exercise18(40,2) */

```

```

        System.out.println("The daily pay in case of overwork is: "+dailyPay);
        return dailyPay;
    }
}

```

**19. There may be some omissions in the `Math` library, e.g.**

- a) cotangent,
- b) secant,
- c) cosecant,
- d)  $\log_2(x)$ ,
- e)  $\log_{10}(x)$

**may be absent. Please write them.**

Java has a predefined class called `Math`. The latter contains a number of standard functions:

```

Math.sin(x);
Math.cos(x);
Math.tan(x);
Math.log(x);
Math.pow(x,y);
Math.sqrt(x);
etc.

```

There exist also two constants defined in the class `Math`:

```

Math.E = 2.718281828..
Math.PI = 3.1415926..

```

Example of the `Math` library usage:

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

is written in Java as

```
x = (- b + Math.sqrt(b * b - 4 * a * c)) / (2 * a);
```

Some functions are not specified in the `Math` library, however they can be easily computed from the existing ones. To make the produced functions easily accessible, it is useful to declare them `public` and `static`.

```

public class double SupplementaryFunctions
{
    public static double cotangent (double radians)
    {
        return 1.0 / Math.tan (radians);
    }
    public static double secant (double radians)
    {
        return 1.0 / Math.cos (radians);
    }
}

```

```

public static double cosecant (double radians)
{
    return 1.0 / Math.sin (radians);
}
public static double log2 (double x)
{
    return Math.log (x) / Math.log (2.0);
}
public static double log10 (double x);
{
    return Math.log (x) / Math.log (10.0);
}
}

```

**A remark on the argument of trigonometric functions.**

Although scientists prefer to work in radians, some engineers find it more practical to work in angle degrees. The transition between degrees and radians may be ensured by the following lines of code:

```

public double sine (double degrees)
{
    double radians = degrees * Math.PI / 180.0;
    return Math.sin ( radians);
}

```

and analogously for other trigonometric functions.

**20. Produce a Java programming model of the exponential growth of the population (the linear regime of the logistic model).**

The logistic model expressed, in the continuous case, by the logistic equation

$$\frac{dp}{dt} = ap - bp^2, p(t_0) = p_0 > 0; a > 0, b > 0 - \text{constants}$$

is a simple model to represent the population growth,  $p = p(t)$ . The meaning of this equation is that the reproductive rate is assumed to be proportional to the number of species or individuals, whereas the mortality rate is proportional to the frequency (probability) of pair encounters (collisions). You can find some material on the logistic model e.g. in [http://www5.in.tum.de/lehre/praktika/comp\\_mod/SS03/MathModeling03.pdf](http://www5.in.tum.de/lehre/praktika/comp_mod/SS03/MathModeling03.pdf)

The logistic equation is a nonlinear ODE; it is simple in the sense that it can be easily integrated. Namely, for  $p \neq \frac{a}{b}$ ,

$$t - t_0 = \int_{p_0}^p dq (aq - bq^2)^{-1}$$

This integral exists only when both  $p$  and  $p_0$  (the actual and initial values of the population) lie in  $(0, a/b)$  or in  $(a/b, \infty)$ . In other words, there exist no solutions that cross the straight line  $a - bp = 0$ . Integration gives

$$t - t_0 = \frac{1}{a} \int_{p_0}^p dq \left( \frac{1}{q} - \frac{b}{a - bq} \right) = \frac{1}{a} \log \frac{p(a - bp_0)}{p_0(a - bp)}$$

Solving this equation with respect to  $p$ , we have

$$p(t) = \frac{ap_0 \exp[a(t-t_0)]}{a - bp_0 + bp_0 \exp[a(t-t_0)]} = \frac{ap_0}{bp_0 + (a - bp_0) \exp[-a(t-t_0)]}$$

One can see that for  $0 < p_0 < a/b$ ,  $p(t)$  is defined for all  $t$ ,  $0 < t < \infty$ , while for  $p_0 > a/b$ ,  $p(t)$  is defined only for

$$t > t_0 - \frac{1}{a} \log \frac{bp_0}{bp_0 - a}$$

In the case  $p_0 = a/b$ , the solution is a constant,  $p(t) = a/b$ , since in this case  $dp/dt = 0$ . One can also see that the solution converges to the constant value  $a/b$ . For  $p_0 < a/b$ ,  $p(t) < a/b$  for all  $t$ , so  $(a - bp(t)) > 0$  and  $dp/dt > 0$ , which means that the function  $p$  (population) monotonously increases. In the opposite case  $p_0 > a/b$ ,  $p(t) > a/b$  for all  $t$  and  $dp/dt < 0$ , which means that the population is monotonously diminishing.

The linear regime of the logistic model corresponds to the case when one can neglect the second term (proportional to  $p^2$ ) in the logistic equation. This is correct for small probabilities of pair collisions and for short observation times. More accurately,

$$\frac{bp_0}{a} (e^{a(t-t_0)} - 1) \ll 1,$$

which gives for the short period of observation,  $a(t - t_0) \ll 1$ , the following restriction on the death rate in the population

$$bp_0(t - t_0) \ll 1.$$

An obvious drawback of the logistic model is that it does not contain spatial variables, which means that it cannot be applied to spatially inhomogeneous situations.

An example of estimation with the help of the logistic model. One may assume the total current (2004) human population to be  $6.2 \cdot 10^9$ , the growth factor to be  $a = 0.029$ , the annual population growth to be  $0.013 \text{ year}^{-1}$  (more or less standard demographic data). Then we have

$$\frac{dp(2004)/dt}{p(2004)} = \frac{d}{dt} \log(p(2004)/p_0) = 0.013 = a - bp(2004) = 0.029 - b \cdot 6.2 \cdot 10^9,$$

which may be considered an equation to find the attenuation factor  $b \approx 2.58 \cdot 10^{-12}$ . The result will be  $a/b \approx 11 \cdot 10^9$ , i.e. the world population tends to converge to approximately 11 billion people. This result is not very sensitive to slight change of the constant  $a$  and of the annual population growth.

Another frequently used form of the logistic equation:

$$\frac{dp}{dt} = ap \left( 1 - \frac{p}{k} \right),$$

i.e.  $b = a/k$ . This form introduces explicitly the asymptotic value  $k = a/b$ . Then the solution may be written as

$$p(t) = k \frac{p_0}{p_0 + (k - p_0) \exp[-a(t - t_0)]}.$$

Now, let us try to produce the Java code corresponding to the logistic model. For simplicity, we shall use the second form (with  $a/b = k$ )

```
import java.lang.Math
public class LogisticModel
{
    // declaration of instance variables
    private long p0; //population at the initial moment t=t0.
    /* Without any loss of generality,we may put t0 = 0, since the model corresponds to
    an autonomous system */
    private double a;
    private double k;

    // Constructor: initialize all the quantities needed for the solution of the logistic
    equation
    public LogisticModel (long initialPopulation, double growthRateA, double MaximumK)
    {
        // initialize instance variables
        p0 = initialPopulation;
        a = growthRateA;
        k = maximumK;
    }

    // Calculate the population at time t
    public long CalculatePopulation (double t)
    {
        // this expression manifests the solution of the logistic equation
        double population = (k*p0) / (p0 + (k - p0)*Math.exp(-a*t));
        // return the population in terms of integer numbers
        return Math.round (population);
    }
}
```

If one would like to insert values for the model from the command line, one can write the following Java application that would calculate the population growth (for simplicity, the linear mode is used here):

```
import java.lang.Math;
public class Population {
    public static void main (String [ ] args) {
        try {
            double p0 = Double.valueOf (args [0]).doubleValue ( );
            double t = Double.valueOf (args [1]).doubleValue ( );
            double a = Double.valueOf (args [2]).doubleValue ( );
            System.out.println(p0 * Math.exp(a * t));
        }
        catch (Exception e) {
            System.out.println ( "Population model: initialPopulation timeYears
growthRate");
        }
    }
}
```

For instance, taking the values  $p_0 = 6.2 \cdot 10^9$ ,  $a = 0.029$ , we get in  $t = 10$  years the world population to become  $p \approx 8.286$  billion people.

One can also use numerical methods (e.g. the Euler's method), instead of analytical solution of the logistic equation, to produce the algorithm and implement it in the Java code. One may as well consider the discrete form of the logistic equation:

$$x_{n+1} = ax_n(1 - x_n/k)$$

## 21. What types of variables in Java do you know? What is a reference type of a variable? Give example(s).

A variable in programming, in distinction to mathematics, is understood as an abstraction that represents a storage location. A Java variable has the following attributes:

**name** – understood as a label used to identify a variable throughout the text of a code.

**type** – determines the set of values a variable can acquire and the set of operations that may be performed on the variable.

**value** – understood as the content of the memory location occupied by that variable (there may be more than one memory locations occupied by a variable). How the contents of the memory locations are interpreted is determined by the type of the variable.

**lifetime** – the time interval in the execution of a Java code, during which the variable exists. Local variables exist as long as the method, inside of which they are declared, is active. Static fields exist as long as the class, in which they are declared, remain processed by the JVM. Non-static fields of a class exist as long as the object, of which they are members, exists.

**scope** – a number of statements defining blocks of code within which the variable is accessible and determining when the variable is created and destroyed (e.g. public, protected, private)

Java is a “strongly typed” language, meaning that each variable has a defined type. The type of a Java variable is either one of the primitive types or of a reference type. Java contains eight simple primitive types, which form four logical groups:

- **Integers:** `byte`, `short`, `int` and `long`,
- **Floating-point numbers:** `float` and `double`,
- **Characters:** `char`
- **Boolean:** `boolean`

Each variable of a primitive type is a direct instance of that type, whereas reference type variables may be thought of as pointers to objects, which, being instances of declared classes, are user-defined types. Any variable, which is not one of the primitive types, is of a reference type.

The difference between the primitive and the reference types can be better understood, if one considers how the respective variables are stored in the computer memory. The primitive type variables are typically stored in stacks (or are “inlined”). A stack is a memory block whose volume may be varied depending on method requirements for the storage of its local variables. When a method is called, the stack grows. When a method is terminated, the stack

shrinks, since the method's local variables are no longer needed. In Java, there exists a **Stack** class that extends the **Vector** class, the latter implementing a growable array of objects. Like an array, it contains components that can be accessed using an integer index. However, the size of a **Vector** can grow or shrink as needed to accommodate adding and removing items after the **Vector** has been created (see e.g. the Java language specification under <http://java.sun.com/j2se/1.4.2/docs/api/java/util/Vector.html>).

Inline storage implies that a primitive type variable is declared as a part of a larger object, like e.g. two point (int x, int y) objects can be inlined into the **ray** object. Inlining also applies to elements in an array. Thus, for primitive types, only values are stored in memory. In this sense, primitive types are sometimes called “value types”.

The storage space allocated for a reference type contains an address – a reference to an object of the appropriate type. This user-defined type is created by defining a class. For instance, by defining a class **Axx**, we immediately introduce a new type that can be used to declare a variable **a**:

```
Axx a;
```

This variable **a** is a reference to an object instance of the class **Axx**.

In Java, instances of a class must be explicitly created, e.g. by the **new** operator:

```
a = new Axx ( );
```

The **new** operator allocates memory space for a new instance **a** of the class **Axx**. The object, once created, remains in memory until the JVM's garbage collector, moving through the entire list of objects, finds out that the object in question is no longer being referenced (the reference count being zero). Then the object is destroyed, with the memory chunk occupied by it being released.

In short, the reference type variable may be interpreted as the address in memory where the object referenced by that variable is stored.

It is possible in Java, for a reference type variable, to refer to nothing. Such references are called null references. One can explicitly assign a null reference to a variable:

```
x = null;
```

One can also test a reference to be null:

```
if (x == null)
    { /* it means that no object is referenced or */ }
```

Example:

A **String** variable is an object of the class **String**. One declares a **String** variable in the way similar to defining variables of primitive types:

```
String myString = "This is a string literal";
/* A constant object of the class String that the compiler creates to use
in the program */
```

This statement declares the variable `myString` of the class `String` and initializes it with the value "This is a string literal". One can store a reference to another string in a `String` variable by using assignment. For instance, one can change the value of the string variable `myString` by the statement:

```
myString = "Strings may be tricky"
```

By executing the assignment statement, the original reference is discarded, the old string is destroyed, and the variable `myString` stores a reference to the new string value.

A similar mechanism of referencing can be applied to arrays. Here again, the array variable is separate from the array itself. Like a primitive type variable can accept different values at different times, one can use an array variable to reference different arrays at different places of the code, for example:

```
double [ ] samples = new double [50]; // an array containing 50 elements of
                                     // the double type
```

It may happen that at a later point of the program a need arises to refer the array variable `samples` to a larger array, with 100 elements. Then one can simply write:

```
double [] samples = new double [100];
```

Now the variable `samples` refers to a new array of 100 elements of type `double` (it can be of type `int`, `float`, `long`, etc., as well as user defined)

## 22. What is the difference, if any, between operators “= =” and “=”?

Operator “=” produces the assignment statement, such as

```
y = x;
```

This statement takes the **value** of the variable `x` (in RHS) and copies that value **into the variable** `y`. After the assignment, `x` and `y` still remain distinct instances that just happen to have equal values. Operator “=” is typically used in Java to initialize variables.

Operator “= =” is the equality operator normally used for a comparison of the form:

```
if (x = = y)
    { /* delivers true if the values contained in the variables x and y are equal.*/}
```

If the “= =” operator is used for reference type variables (objects), e.g.

```
if (F = =G)
    { /* tests whether F and G refer to the same object instances */}
```

the value **true** is returned only in case `F` and `G` point to the same object.

If `F` and `G` refer to distinct object instances that happen to be equal, the test still fails. To test whether two distinct objects are equal, it is necessary to call the `equals` method:

```

if (F.equals (G))
{ /* returns true if F and G are the same object, not just equal; returns false if they are
different objects, even if they have identical values in their data members */}

```

**23. Write a Java program to compute the sum**  $S(x, n) = \sum_{k=0}^n x^k$  .

Despite an apparent simplicity, there may be several ways to solve this problem in Java. The simplest code is the following:

```

public class Sum
{
public static double Series (double x, int n) /**it can as well be of int or float types
* in principle, "static" is not necessary
* here; the method may be non-static*/
{
double sum = 0;
for (int i = 0; i <=n; i++)
{
double temp = 1;
for (int j = 0; j < i; j++)
temp *= x;
sum += temp;
}
return sum;
}
}

```

A more elegant solution:

```

public class Sum{
// Constructor for objects of the class Sum - not necessary, if the new operator is
// called later, see below
public Sum(){ }

// The method of calculation
public double Series (double x, int n)
{
double sum = 1;
double temp = 1;
for (int i = 1; i <=n; i++)
{
temp *= x;
sum += temp;
}
return sum;
}
// To save writing, one can include a recursive version in the same code
public double Series_Recursive(double x, double temp, int n)
{

```

```

        if(n>0)
            return temp+Series_Recursive(x,x*temp, n-1);
        else
            return temp;
    }
// To test the code, let us write the main method
public static void main(String [ ] args)
{
    Sum sum = new Sum();
    System.out.println(sum.Series(2,4));
    System.out.println(sum.Series_Recursive(2, 1, 4));
}
}

```

Another variant would be to program the mathematical expression resulting from summation:

$$\text{Sum} := S(x, n) = \frac{1-x^{n+1}}{1-x},$$

where the argument  $x \neq 1$  may be called “base” and the exponent  $n$  may be called “limit”.

```

public class SumDirect
{
    // Constructor – not necessary, if the new operator is called later
    public SumDirect ()
    {
    }
    // Declaration of methods
    public double Series (double x, int n )
    {
        double result;
        if ( x ==1)
        {
            result = n++;
        }
        else
        {
            result = (Math.pow (x, n +1)-1) / (x -1);
        }
        return result;
    }
    // the main method
    public static void main (String [ ] args) {

        /* Let us provide the opportunity to input values from the command line

        double x = Double.valueOf(args[0]).doubleValue();
        int n = Integer.valueOf(args[1]).intValue();
        double result;
            SumDirect sumdirect= new SumDirect();
        result = sumdirect.Series (x,n);
        System.out.println("Sum = " + result);

```

```
}  
}
```

A very simple variant would be:

```
public class SumDirect1  
{  
    // Declaration of fields  
    private double x;  
    private int n;  
  
    // Constructors  
    public SumDirect1 (double base, int limit)  
    {  
        x = base;  
        n = limit;  
    }  
  
    // Declaration of methods  
    public void Series ( )  
    {  
        if (x == 1)  
        {  
            System.out.println("Sum = " + n++);  
        }  
        else  
        {  
            double result = (Math.pow (x, n + 1) - 1) / (x - 1);  
            System.out.println("Sum = " + result);  
        }  
    }  
}
```

**24. Write a Java program giving a solution of a second-degree (quadratic) equation. Test your program on the following equations:**

$$x^2 - 4x + 8 = 0;$$

$$x^2 - 4x - 12 = 0;$$

$$2x^2 + 13x + 21 = 0.$$

**Of course, you may produce your own examples of second-degree polynomials.**

```
import java.math.*;  
public class QuadraticEquation  
{  
    // Declaration of variables  
    private double _a;  
    private double _b;  
    private double _c;
```

```

public double D;

// One can declare an array here, of course

private double x1, x2;

// Constructor for objects of the QuadraticEquation class
// Initialization of the equation coefficients – instance variables
public QuadraticEquation (double a, double b, double c)
{
    _a = a;
    _b = b;
    _c = c;
}
// The solution method
public void solveEquation ( )
{
    double D = Math.pow(_b,2) - 4 * _a * _c;
    if ( _a == 0)
    {
        double x = -_c/_b;
        System.out.println("The equation degenerates into the first-
degree equation, one root:"+x);
    }
    if (D == 0)
    {
        // case 1: two confluent roots
        double root = (-_b) / (2*_a);
        System.out.println("2 equal roots; x1 = x2 "+ root);
    }
    else if (D > 0)
    {
        //case 2: two real distinct roots
        double x1 = (-_b + Math.sqrt(D))/(2*_a);
        double x2 = (-_b - Math.sqrt(D))/(2*_a);
        System.out.println("2 distinct real roots " + x1 +" "+ x2);
    }
    else
    {
        /* case 3: D < 0: 2 distinct complex roots, x1 = u +iv; x2 = u – iv, where
        u = -b/(2*a), v = sqrt (-D)/2*a, i.e. u is declared to store the real
part,
v is declared to store the imaginary part */
        // get the real and imaginary parts
        double realPart = (-_b)/(2*_a);
        double imagPart = (Math.sqrt (-D)) / (2*_a);
        System.out.println ("2 distinct complex roots: x1 = " +realPart + " + i"+
imagPart);
        System.out.println (" x2 = " +realPart + " - i"+ imagPart);
    }
}

```

```

    }
    public static void main (String [ ] args)
    {

        QuadraticEquation _a = new QuadraticEquation (1,-4,8);
        _a.solveEquation();
        QuadraticEquation _b = new QuadraticEquation (1,-4,-12);
        _b.solveEquation();
        QuadraticEquation _c = new QuadraticEquation (2,13,21);
        _c.solveEquation();
    }
}

```

## 25. How can you represent complex numbers in Java?

Unfortunately, Java does not explicitly support complex numbers. However, complex arithmetic is extensively used in scientific computations, and it is highly desirable that addressing complex values and getting access to elements of complex arrays should be as direct as the manipulation of primitive data types like `float` and `double`. From a mathematical viewpoint, a complex number is composed of two parts: a real part  $u$  and an imaginary part  $v$ . There are three main representations of complex numbers: by real and imaginary parts in Cartesian coordinates, by vector radius and angle in polar coordinates, and by amplitude and phase in exponential form. The transition between these manners of representation is straightforward. When creating a class of complex numbers, one must ensure that the objects we are going to create should behave like numbers, i.e. they can be added, multiplied, subtracted and divided. Besides, one must be able to get their amplitude and phase.

```

import java.lang.Math;
public class Complex
{
    public double u;
    public double v;

    public Complex (double re, double im)
    {
        u = re;
        v = im;
    }
    public double Real ( )
    {
        return u;
    }
    public double Imaginary ( )
    {
        return v;
    }
    public double Magnitude ( )
    {
        return Math.sqrt(u*u + v*v);
    }
    public double Arg ( )

```

```

    {
    return Math.atan (v/u);
    }
    public Complex plus (Complex z)
    {
    /* the result should be also a complex number, so we must introduce another
    variable of the type Complex *. This method is different from others in the same
    code, and it is included only to demonstrate the alternative/
    Complex sum = new Complex (u, v);
    sum.u = u + z.u;
    sum.v = v + z.v;
    return sum;
    }
    public Complex minus (Complex z)
    //strictly speaking, this method is a specific case of Complex plus
    {
    return new Complex (u - z.u, v - z.v);
    }
    public Complex times (Complex z)
    {
    return new Complex (u*z.u -v*z.v, u*z.v + v*z.u);
    }
    public Complex divideBy (Complex z)
    {
    double r = z.Magnitude ( );
    return new Complex( (u * z.u + v * z.v)/ (r * r), (v * z.u - u * z.v) / (r * r));
    }
}

```