JAVA FOR THE STUDY OF EVOLUTION

VOLUME VI

StatTrek

Our Basic Statistics toolbox.

José del Carmen Rodríguez Santamaría
Edited and commented by: Write here your name *
The EvolJava Community.
* Other affiliation

Original source: http://evoljava.com/

Contents

1 Starting up **1**
- 1.1 General terminology . . . 1
- 1.2 Installing Java and Eclipse . . . 2
- 1.3 Purposes and projects . . . 4
- 1.4 Our first program . . . 5
- 1.5 Bugs and debugging . . . 7
- 1.6 How to import the eJVol6 project . . . 9
- 1.7 Review . . . 9

2 First version of the toolbox **11**
- 2.1 Mean and var of a list of data . . . 11
- 2.2 Grouping . . . 27
- 2.3 Toolboxes . . . 33
- 2.4 Conclusion . . . 51

3 The binomial distribution **53**
- 3.1 Patchwork style . . . 53
- 3.2 Conclusion . . . 57

4 The normal and associated distributions **59**
- 4.1 The z distribution . . . 59
- 4.2 p-values . . . 60
- 4.3 Merging pieces into a unity . . . 63
- 4.4 Critical values . . . 88
- 4.5 Conclusion . . . 97

5 The dictionary of experiments **99**
- 5.1 The simplest experiment . . . 99
- 5.2 Popular experiments . . . 157
- 5.3 Conclusion . . . 161

6 Conclusions **163**

Answers to exercises **167**

Codex

Programming languages have at presently a very tight syntax, and Java is not an exception. So one must learn the formalism used for each case. We can find here the most usual expressions. A program that contains the referred instruction is also indicated.

9 pag 6. Printing a message to the console:
`System.out.println("I` am starting up! ”);

18 pag 12. Type `int` and `double`:
`int`i = 4; `double`r = 1.234;
18 pag 12. Printing the value of variable x:
`System.out.println("`Value of x = ” + x);

18 pag 12. Declaration of a procedure:
private static double sumVector(double[] Vect)

18 pag 12. Declaration of an unidimensional array or vector:
double DataVect[]={5, 3, 4.5, 2};

Declaration of an array with integer entries

```
//Declaration
int data[];  // data is the name.
data = new int[ 10 ]; //memory for 10 entries
//Assignation
Data[2] = 3;
int n = Data.length; //the length of the array Data.
```

22 pag 14. Declaration of a String

```
String a = "Hello";
```

The Tabulator: "\t"Concatenation of Strings:

```
String s4 = s1+s2+s3;
```

28 pag 18. The for structure:

```
for ( int i = 0; i < n; i++ )
Vect2[i] =  Vect[i]*Vect[i];
```

31 pag 20. Objects in OOP (Object Oriented Programming). Go to the text.

39 pag 28. Declaration and assignation of a two dimensional array

```
int  Data[][] = {{3, 9},
       {4, 10},
       {5, 11},
       {6, 12},
       {7, 13},
       {8, 14}};
```

To address specific entries in a two dimensional array, we must keep in mind that Java begins counting form zero:

```
int z = Data[3][1]
```

In general, the first index address the row and the second the column.

62 pag 57. The while structure

```
while (this condition meets)
{
  do this and that;
}
```

63 pag 57. The if-else structure: REMEMBER THE DOUBLE ==. Logic: the and operator &, the or ||, the negation !.

```
if ( (Barriers[j] <= Vect[i]  ) & (Vect[i] < Barriers[j+1]) )
  FreqTable[j][1] =FreqTable[j][1] +1;
```

The negation of a Boolean clause:

```
if (!(i==2)) |(!(j==2))
```

72 pag 64. The switch structure that is equivalent to many ifs in parallel.

```
       switch (k)
{
case 0: zero(); break;
case 1: one(); break;
case 2: two(); break;
case 3: three(); break;
case 4: four(); break;
case 5: five(); break;
case 6: six(); break;
case 7: seven() ; break;
case 8: eight(); break;
case 9: nine(); break;
default: other(); break;
}
```

78 pag 68. Abtract classes: go to the text.

Some other useful instructions:
The Math API

```
 double r = 5.2
 doube rSquare = Math.pow(r,2);
 double rCube = Math.pow(r, 3);
 double rEightRoot = Math.pow(r, 0.125);
double rSquareRoot = Math.sqrt(r);
```

Change of type `double` to `int`:

```
double aa = 3.4;
int i = (int) aa;
System.out.println(i);
```

Random numbers

```
import java.util.Random; //At the preamble of the program
Random r  = new Random(); // somewhere at the beginning

//A random number from 0 to 5
r.nextInt(6)   //In within a method

//Other variants
Random k = new Random(456); //seeded mode for exact repetitions.
r.nextLong();
r.nextFloat();
r.nextDouble();
r.nextGaussian();//normal with mean 0 and dev 1.
```

Random numbers that obey a normal random variable with predetermined mean equal to μ and deviation σ:

```
double d = r.nextGaussian()
int c = (int) (Math.floor(sigma*d + mu));
```

Random numbers and chars

```
import java.util.Random; //At the beginning of the program
Random r  = new Random(); // somewhere at the beginning

//A random number from 1 to 5
r.nextInt(6)   //In within a method

//This is a char or letter
char n = 'j';

//A random upper case char:
char ch =   ( char ) ( r.nextInt( 26 ) + 65 );

//A random lower case char:
ch =  ( char ) ( r.nextInt( 26 ) + 97 );

//Other variants
Random r = new Random (seedValue);

r.nextLong();

r.nextFloat();

r.nextDouble();

r.nextGaussian();
```

Operations with strings that are necessary to simulate evolution. The Replace operator:

```
s2 = s1.replace("ion", "ate");
c.replace(a,b);
```

Conversion of number to string. If the number is of type double, we use:

```
   s1  = String.valueOf(r2);
```

When the number is of type int, we use the next procedure:

```
int i = 3;
Integer n = i;
String s = n.toString();
System.out.println(s);
```

Copy operator

```
s2 = s1.substring(4);
s3 = s1.substring(4,8);
```

The length of a String:
`int` n = s1.length();
Char c is converted into string s

```
String s = c.toString();
```

A number from 0 to 9 is converted to a char.

```
//Integer number i is converted to char z,
//when z is printed it looks just like i.
char z=  ( char ) ( i +48  );
```

A char is withdrawn from a String s and a char is given a numerical code:

```
 char c  =   s.charAt(0);
int sign = Character.getNumericValue(c);
```

Preface

The series *Java for the Study of Evolution* is directed to scientists that want to manage a serious but not excessively expensive tool to study evolution by direct experimentation under perfectly controlled conditions. These requirements cannot be met in nature but only in simulations and mathematical models. In consequence, the series has three main purposes:

1. To endow the community of **researchers in biology and evolution** with *high level programming*, enabling an accurate study of models and simulations of the most diverse nature.

2. To clearly show how this tool is used to study the fundamental questions of evolution.

3. To suggest that the study of Java could be *very fruitful* for **undergraduates** in biological sciences even more than a course centered at calculus.

This Volume VI is dedicated to accompany those persons that already worked Vol V, the scientific method with Java, and pursue a serious improvement in their Java skills by developing a Java statistical toolbox. The final product, StatTrek, is our toolbox of programs specially targeted to support learners and teachers of basic statistics. While we work, we will have the opportunity to witness from first hand how the evolution of software that is developed by human beings in a concrete case is.

Every part of all our texts and programs in whatever volume can be edited, commented and criticized. Because every writing implies a responsibility, make explicit at a frontal place whether or not the original material was modified and by whom. Next, give a link to the original source and follow your own style and ethics. For instance, your own comments might be included in special paragraphs entitled as *comment*.

Bogotá, Colombia, *José Rodríguez*
June 2011

Introduction

In spite of all marvelous statistical packages at disposition, one always longs for a program to fit exactly in personal needs. The solution is simple: do it by yourself. So, we share here a basic Java material that everyone can adapt to his or her personal directives. The material is guided by our companion volume on *Basic statistics* and *Vol V, The scientific method with Java*. This material is addressed in first instance to persons that already have worked volume V, the scientific method with Java, and want to compare his or her personal experience of making a small project, to devise a toolbox for statistics, with that of other developers. Given that we continue the work began in Vol V, we will reuse as much material of that volume as possible.

On the other hand, some people would like to just use the toolbox without worrying too much about learning Java or the scientific method. They are welcome and so we append a first chapter on how to begin from scratch to run a Java program and how to carry tiny modifications to run own data and next they are addressed to the answer of the last exercise of the last chapter. If some of those persons fall into the temptation of learning more Java, they are invited to work over the first chapters of the aforementioned Vol V of the series Java for the study of evolution.

One might use this material to :

- Verify own answers to exercises in statistics.
- Swiftly calculate answers to problems for the redaction of, say, personalized examinations.
- Pursue dexterity in Java programming. In fact, this project is the answer of the Author to a challenge that was proposed in Vol V, the scientific method with Java. So, Java learners can study this material to guide themselves in their own work.
- Learn more about evolution: any software project is an experiment in evolution, i.e., in the modification of software.
- Develop personal projects.

The title of our enterprise, StatTrek, points to two facts. First, the path of programming is a very difficult and long. Second, keeping alive the feeling of adventure, of improvising, of pursuing dreams is the trademark of creativity.

Chapter 1

Starting up

```
Beginning from total ignorance.
```

1. ***Purpose.*** *We learn the general terminology, how to acquire Java and Eclipse, how to install them, and how to run a first program.*

1.1 General terminology

There are some usual terms that we specify below.

2. ♣ ***General definitions***

In first place, let us notice that we, human beings, use languages to communicate one with another. A **language** is composed of verbal instructions, a code of interpretation and a semantic content. A **verbal instruction** is a string of sounds or letters that belong in an **alphabet**. Verbal instructions must be deciphered according to a **code of interpretation**, which appears, say, in a dictionary. The deciphered verbal instructions convey information about the world or about actions that must be undertaken and it is in this way that they acquire **semantic content**.

A **computer** is a devise that has the potential ability of executing verbal instructions. A **program** is a set of verbal instructions, written in a specific **programming language** that can be interpreted and executed by a computer. **Code** is a set of instructions that eventually contain correctly written programs. **Software** is a set of programs that execute a task and very specially that create a friendly interface with the human. **Hardware** is the wiring that converts programs into specific actions. A **robot** is a machine that is commanded by a computer that has been programmed to have certain independence.

A **developer** is an entity, a person or a process, that is committed to the design of software, i.e., of programs with a specific function, which is defined before beginning but that maybe modified along the process. It is absolutely sure that the first trial of a code is unviable. But if one manages to produce a

viable code, most surely it will not do what one was so eagerly expecting. The process of adjusting the code to the predefined purpose is called **debugging**.

Our study of **Java**, our platform of software design, is specifically directed to show that the insight that **the genome is software and evolution is a software developer is both necessary and sufficient to deal with all fundamental questions about biological evolution**.

To use Java directly is not as easy as using it in duo with Eclipse, which is a precious interface of Java with the developer. All what follows refers to this ensemble that is generically referred to as Java. In strict sense, **Java is a great project** whose grandiosity might overwhelm at anytime.

The name Java makes reference to a family of ants that lives in the island of Java. In a single tree but in different leaves one might find various nests of the same colony. The colony has various queens in proportion to the number of nests, which also determines the number of adult workers and immatures. But the numbers of males and of new queens go a path on their own.

3. ♣ ***Definitions.*** *A* ***method*** *in Java is a portion of code that is designed to fill a determined function. A set of methods maybe gathered in a* ***class*** *to execute a great task. A class must be saved in a file with the same name, a relation that is case-sensitive. Classes are organized in books that are called* ***projects****.*

We will learn to install **Java**, to learn how to use **Eclipse** to run an existing Java program, and to manage the fundamental ingredients of programming that are directed to study models and evolution.

The basic items to simulate **evolution** are strings and numbers. That is so because the DNA is a string of characters whose alphabet is A G T C, while many variables of phenotypic interest are naturally encoded in the form of numbers.

1. ***Comment.*** *You can edit this work in whatever way you want. To insert your own annotations, criticisms, doubts, open questions, mathematical proofs you can insert a comment, such as the present one, which might be cited as follows: comment 1 on page 2.*

1.2 Installing Java and Eclipse

Sun Microsystems composed Java as a great symphony and distributed it without fees. Sun Microsistems was acquired by Oracle in 2010, which continues its free distribution. Use Google to contact this company and to load and install Java on your desktop. To that aim, you can prompt Google with `java download` or you can go in through a direct link (Oracle, 2011a).

Java comes with a lot of tools, which together form a run time environment.

The web is full in tutorials about Java. As examples, it would be nice to give a look at the material offered by Oracle (2011b), or by Fred Swartz (2007) or by Developer (2011).

Follow the instructions to install Java and forget about it because Java is accessed through an interface, **Eclipse**, which is also freely distributed. Please, use Google again to localize Eclipse, next load and install it. Else, contact Eclipse directly (Eclipse, 2011). The book of Holzner (2004) is of great help to acquire a global vision about this interface but Eclipse itself has evolved and it may be preferable to study the guides given by the application itself.
Eclipse offers various options for downloading. You can try

Eclipse Classic.

While installing Eclipse, it demands to create a special file that is known as **workspace** and that is used by Eclipse for its ordinary work. It is possible to create various workspaces, with the name one wants, and to change the used one.

The first task is to play with the new toy. If you want to be a great master, you must learn to enjoy playing permanently. By the way, give a try to your kids: most surely, they are looking for something really cool. Incite them to look here and there and try out whatever comes to the mind.
The first window that Eclipse opens for you is the WELCOME window. In *Tutorials* → *Create a Hello world application*, follow the instructions and try to create your first program. If you fail, you might try again. To that end, you can rescue the Welcome window from the help at the main menu of ECLIPSE. Else, use our methodology, which comes below.

4. ***Guarantee.*** *If after some two hours of work you cannot follow the instructions of the present chapter to install Java and Eclipse and to run the first program, consult a helping tutorial video by Dexter(2011). If this is of no help, it is mandatory that you write me because you are guaranteed to have a happy start. Furthermore, we present many programs that are guaranteed to work perfectly on Java Runtime Environment java-1.6.0-openjdk1.6 and Eclipse SDK 3.4.0. Memory rarely might become important: 2Gb of RAM migth be sufficient for most cases and 8 was enough for all our instances.*

Please, express your discomfort to

jose@evoljava.com

You can find more material at

http://www.evoljava.com

It is rather difficult to understand how a program is designed but we have made an effort to explain why each program works, and the reader is committed to learn how to modify them to create variations that fulfill slightly modified functions. So, the first duty is to learn how to copy a program into Eclipse that it could be run. The first step is opening a project.

1.3 Purposes and projects

In Java, before copying or designing a program, one must make clear what the purpose one has in mind is. Our present purpose could be just to begin. This intention defines a project: Project6. Then, we must communicate to Java our intention.

5. ***Opening a project***

To open a **project**, please, follow the sequence of options beginning from the main menu: $File \rightarrow new \rightarrow project \rightarrow Java \rightarrow Project \rightarrow Next$. A window will pop up for you with an invitation to fill the relevant data about your project. Give it a name: *Project6*. Click Next. In the new window, click Finish.

Now that we have a purpose, a project, we can learn to append a program to our project into Eclipse, the interface of Java with the user.

6. ***Appending a program***

The first thing we must do is to acquire the right to append a program into a project. In Java a program is called class. So, let us click on $File \rightarrow new \rightarrow$ *class* and a window will pop up. Let us fill in the name of the source project: Project6 (else browse for it). Let us give a name to our new class: *console*. If desired, we could activate the *Public static void main* property and then we may click on *Finish*. Next, one shall see a new window with the name *console.java* in its upper bar. We can edit our new program in that window.

7. ***Copying a program***

Programs are not typed except when they are created. Otherwise, they are transcribed by a copy-paste procedure, for which you have various options:

1. To copy each program from the present pdf document. In this case, you must resolve a little problem: the copy operation also copies navigating signals of the document, headers or page numbers, so they must be identified and deleted. The identification is done automatically by Eclipse after the paste operation is complete: extraneous lines of text appear with a red mark at the right margin. They must be deleted.

2. To get the LaTeXsource, which can be opened with any text processor. You can copy any program from it without suffer anyhow. Additionally,

you are invited to use our LaTeXsource to devise your own material for yourself or for your students. In general, feel free to do anything good for forming a healthy and strong **evolJava** community.

3. To get a file, in plain text, with all programs from which you can copy the desired program immediately. To get the file, go to http://www.evoljava.com

The copy-paste procedure might consists of the following steps:

1. Clear from the editor or main window of Eclipse the code that was automatically written by Eclipse for you, if any.
2. Highlight the text that you want to copy.
3. Copy the highlighted text to the clipboard.
4. Verify that the active window corresponds to the chosen program, *console* in this case. To activate a window with a program, click on its name in the upper bar of the editor window.
5. Paste the enclosure of the clipboard to the blanked document. Warning: Eclipse for Windows is not based on Windows but on DOS: the command for **delete** is control + x, for **copy** is control + c, for **pasting** is control + v, for **select all** is control + a.
6. Save the new program: click on the corresponding icon.

1.4 Our first program

Programs must obey a certain pattern or structure that is complexity dependent.

8. ***The simplest structure***

Our first programs will have the next structure:

```
//Program F8 structure
//Same as Program   A0
//This is the structure of a program.
//The name in Java of this program is structure
public class structure
{
  //Every program has a main
  public static void main(String[] args)
   {
    //Here are the instructions
   }
}
```

When Java runs a program, it looks in first instance for the main method and begins its execution.

We have below a true program whose purpose is to write a message in the console. The **console** is a special window where Eclipse reports results and errors. If whatever cause hides the console, one can restore its visibility in any one of the two following options:

a) In the main menu follow the path: $Windows \rightarrow ShowView \rightarrow Console$.

b) Click Alt+Shift + Q, then click C.

9. *Printing in the console.* *The next code contains a program, whose purpose is to print a message to the console.*

```
//Program F9 console
//Same as Program   A7
//This is my first program.
//The name of this class is console because
//it prints a message to the console.

public class console
{

 //The next is the main method
 public static void main(String[] args)
 {

  // This is a comment.

  // Comments help to understand programs.

  // The next line contains an instruction
  // that prints a message to the console:

  System.out.println("I am starting up! ");

 }
}
```

We use in this program the method **System.out.println**. This method prints a message to the console and posits the console cursor at the beginning of the next line. One can use 'print' to write something but remaining in the same line.

10. ***Exercise*** *Observe in the anterior code that parentheses and keys always come by pairs. Java allows placing each parenthesis with freedom. Our directive*

is to enable a quick visualization of blocks. Point to the lines that contain a comment and to those that contain code properly, i.e., instructions that Java must decipher and execute. What distinguishes code from commentaries? Which is the function of the semicolon?

Compare your answers with those of the Author that exist for most exercises. They appear in the chapter *Answers*. Exercises are part of the text and results might be used in the sequel.

11. ***Running our program***

To run a program or class, do the following:

1. Right-click on the editor window that contains the text of the new class.
2. In the sub-menu that popped up, click on *Run as* → *Java* → *Application*.

In response, we shall see the output of our first program that writes in the console the phrase: *I am starting up!*

12. ***License.*** *Dear Reader: as part of the exercises, you must modify the programs. So, you have the* ***license*** *to use all programs written in this book in whatever form you want even to get rich and very specially to modify them as you are asked in the exercises. This freedom might help the formation of a strong EvolJava community.*

Let us do our first exercise in varying a program. We are artificially evolving software.

13. ***Exercise*** *Modify our first program in such a way as to write in the console your name and birthday.*

1.5 Bugs and debugging

The most immediate reality of every developer, no matter how good and skilled he or she might be, is the great quantity of errors that he or she commits when composing a program or modifying a piece of software.

14. ***A program is known by its fruits***

There are two types of errors: those that hinder compilation and those that hinder function. **Compilation** is the process of translating a program that is written in Java into machine language, which is composed of elementary instructions that can be executed by the computing hardware.

A compilation error or **bug** happens when the program is not accepted by the compiler. Eclipse is very gentle with the developer and it marks errors in red. To **correct an error**, drag the cursor to the error and wait a moment while Java pops up a message with the explanation of the nature of the error and possibly with a suggestion to correct it. Errors are also marked at the

navigating vertical bars that surround the editor window. If in the left bar a red X appears, drag the cursor to it and an explicative message will appear. If a yellow light also appears, right click on it to read more suggestions for error correction. A suggestion can be executed at once by double clicking on it.

Sometimes a program is not accepted but it has no errors. This happens when one follows the procedure to run a program: *right clicking* → *Run as* but nothing is given to choose. In this case one has a grammatical or syntactic error, say, the main method is not declared as *static*.

The most difficult errors happen when the program is accepted for compilation, runs and produces results but these make no sense in regard with the intended function of the program. To correct these type of errors, one must keep in mind that software is in general highly informative and so to know its output there is only one way, to run the program in the computer or in the mind, to see what it does. In general, mental calculations are not reliable and so one must resort to real execution: **software is known by its fruits**. This means that one must force the program to write out intermediate results to follow detailed calculations. To do this, one inserts lines of code with instructions to write over the console interesting variables and their names. Next, one compares output with correctly expected results and can decide whether or not the error in design has been committed before of after the line under scrutiny.

Eclipse has special facilities for **debugging**, which is the process of cleaning bugs. To this aim, it is good to activate before everything else line numbering: one must right click over the left vertical bar that surround the editor window. A menu will pop up and over it, one checks the box corresponding to *Show line numbers*. Next one must target a given line of the project as compilation breaking point: right click over the left vertical bar of the editor window and choose *Toggle Breakpoint*. Next, follow the menus *Window* → *Open perspective* → *Java Browsing*. Next one chooses *Run* → *Debug as* → *Java Application.* The program will run until the breakpoint and a table will be displayed with the variables and their names when the execution runs at this precise place. Additionally, helping menus get activated: they are located in the Menu *Run* → *step in, step over*. Play a bit with them to see what happens.

Reality seems to recurrently show that apart from never stop suffering there is nothing else to debugging. Nevertheless, some people consider that debugging is a science on their own, so, they develop black lists with statistics and special strategies for catching bugs and things like that. Actually, there are programing languages and styles of developing software that are error prone. We will discuss below some strategies directed to attack such weaknesses, which can not be completely abolished because bug free software is technically an impossibility. The reason is that software to be bug free must be tested for every possible case and this can take longer than allowed time.

My first black list of bugs:

1. Terminal semicolon ";" is lacked.

2. A variable has been erroneously typed.
3. Keys or parentheses do not go by pairs.
4. The type of a variable has been violated, say, variable `i` has been declared as int (integer 0, ±1, ±2, ...) but it is instantiated as `int i = 3.56;`

1.6 How to import the eJVol6 project

Eclipse offers the opportunity to effortlessly import complete projects with all its files. All one needs is to turn Eclipse with its already created workspace and follow the next procedure.

15. *The eJVol6 project*

Let us import into the workspace the *eJVol6* project with all its ready to be run files. To that aim:

1. Download the *eJVol6.zip* file associated to Vol 5 and keep it in a suitable folder.
2. Over Eclipse follow the menus: *File* → *Import* → *General* → *Existing projects into Workspace* → *Select Archive File*. In the new dialog, browse for the folder where you put the file. Click over the file and next on *Finish*. This operations create a new project, *eJVol6*, with all the source files of Vol V.
3. To see your recently created project, go to the menu *Window* → *show view* → *Package explorer*. On the corresponding window, look for your project and expand its tree to find source files. After that, all Java files will appear below *src*.
4. To run a specific program, select it and next pulse F3 to show it on the *editor window*. Left click on the editor window and with right clicking follow the *run as* menu. Next, choose *Java application*.
5. To see the output of the program, go to *Window* → *Show View* → *Console*.

These instructions are guaranteed to function. Otherwise, write me to negotiate a truly functional set of instructions.

1.7 Review

We have learned how to acquire and install Java and its companion Eclipse and we have created our first project to which we appended our first program.

Chapter 2

First version of the toolbox

`Like a grain of mustard`

Along the next chapters, we will recurrently work out more powerful implementations of our toolbox, so we will have the opportunity to witness from first hand how the evolution towards perfection is. We will reuse developed material in Vol V as much as possible. People interested in just using the toolbox can work this chapter to next pass directly to the final version of the toolbox which is presented in the answer to the last exercise.

16. ***Purpose.*** *We build a toolbox to calculate the mean and variance of data in diverse formats.*

2.1 Mean and var of a list of data

17. ***Definitions.*** *Data are said to be* ***quantitative or numeric*** *when they are numbers for which arithmetic operations make sense. If we have a* ***list*** *of numerical data X and a given datum is x_i, meaning that it appears in place i, the* ***mean or average*** *is*

$$\bar{X} = \frac{\Sigma x_i}{n} = \frac{\Sigma x}{n}$$

Sample variance *s^2: measures the nonconformity of the data if they are represented by the mean. (We divide by n in descriptive statistics and by $n-1$ in inferential tasks.)*

$$s^2 = \frac{\Sigma (x_i - \bar{X})^2}{n-1} = \frac{\Sigma (x - \bar{X})^2}{n-1}$$

The ***sample standard deviation*** *s is the square root of the variance:*

$$s = \sqrt{\frac{\Sigma (x_i - \bar{X})^2}{n-1}} = \sqrt{\frac{\Sigma (x - \bar{X})^2}{n-1}}$$

***Coefficient of variation C**: measures the degree of homogeneity of data with respect to the mean*

$C = \frac{s}{X}$

Other people prefer the coefficient of variation to be given in percentage.

18. ***The code for the mean and variance follows.***

```
//Program F18 MeanVarList1
//This program computes the mean and
//variance of a list of data.

public class meanVarList1
{

//Prints data in vector
private static void printVector(double Vect[])
 {
  //System.out.println( "Data are" );
  for ( int i = 0; i < Vect.length; i++ )
   System.out.println( Vect[ i ]  );
 }

 //Sums data in vector with real numbers
 private static double sumVector(double Vect[])
  {
   int n = Vect.length;
   double sum = 0;
   for ( int i = 0; i < n; i++ )
  sum = sum + Vect[i];
   return sum;
 }

//The mean of a list of data
private static double meanVector(double Vect[])
 {
  double sum = sumVector(Vect);
  int n = Vect.length;
  double mean = sum / n;
  return mean;
 }

//The variance of a list of data
private static double varVector(double Vect[])
```

```
 {
  //Mean
  double mean = meanVector(Vect);
  // Variance is calculated by definition
  int n = Vect.length;
  double sum2 = 0;
  for(int i = 0; i < n; i++ )
   sum2 = sum2 + (Vect[i]-mean)*(Vect[i]-mean);
  double var = sum2 / ( n-1);
  return var;
}

//The mean and variance of a list of data
private static void meanVarVector(double DataVect[])
{
System.out.println("\nNumber of data = " + DataVect.length);
double mean = meanVector(DataVect);
System.out.println( "Mean = "+ mean);
double var = varVector(DataVect);
System.out.println( "Variance = "+ var);
double deviation = Math.pow( var, 0.5);
System.out.println( "Deviation = "+ deviation);
double coeffVar =  deviation/mean;
System.out.println( "Coefficient of variation = "
                                  + coeffVar);
System.out.println( "Coeff around 0.01, " +
      "homogeneous population.");
System.out.println( "Coeff around 0.1, " +
      " +/- homogeneous population.");
System.out.println( "Coeff around 0.5, " +
      "  heterogenous population.");
}



//Mean and Variance of a list of data
public static void meanVarVector( )
{
System.out.println( "MEAN AND VAR OF A LIST OF DATA");
//Example 19 page 9.
double  DataVect[]  = {5, 3, 4.5, 2};
printVector(DataVect);
meanVarVector(DataVect);
}
```

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

}
```

19. ***Self-test.*** *The list of data 1,1,1,1,1 must produce mean 1 and variance 0. Since these results are obvious, one can use them to test our program. A test of this sort is called a* ***self-test****. One also can use examples of the preferred text. To carry out a self-test is the very first duty of every developer over every program. To make the auto-test, the Author used examples of the companion book Statistics Volume I, version 7. Reference number and page are cited in within the program.*

20. ***Exercise.*** *Run the program and play with the code, i.e., make a self-test and process your own data.*

21. ***Displaying tables.*** *Java is enabled with a powerful formalism to write tables that are both pleasant to look at and with the precision one desires. One has two equivalent methods:* `printf` *and* `format`*. We will use the first, which means, print with format.*

22. ***The code for a table that reports squares under various formats****.*

```
//Program F22 table
//The printf instruction is used
//to write tables.
public class table {

 public static void main(String[] args)
  {
//Printing numbers of type int
for(int i = 0; i < 12; i++)
 System.out.printf( "%7d %12d %n", i, i*i);
//Printing numbers of type double
for(int i = 0; i < 12; i++)
 {
   double a = i*i;
System.out.printf( "%7d  %11.5f  %n", i, a);
 }
//Markers are inserted
for(int i = 0; i < 10; i++)
 {
   double a = i*i;
```

```
System.out.printf( "     (%d) %11.2f %n", i, a);
 }
for(int i = 10; i < 12; i++)
 {
   double a = i*i;
System.out.printf( "    (%d) %11.2f %n", i, a);
 }
//Printing strings
String s = "abcdefghijklmnopqrstuvxywz";
System.out.printf( "%1s    %n",  s);
System.out.printf( "%10s   %n",  s);
System.out.printf( "%40s   %n",  s);
  }
}
```

23. ***Exercise.*** *Run the program and play with the code. Describe how the* `printf` *method functions. Overshadow an appropriate region of the output and paste it to your preferred worksheet and draw the corresponding parable. In SpeadSheet of LibreOffice for OpenSuse: copy to clipboard → paste special → unformatted text → Separator options = space + merge delimeters → chart → XY(scatter).*

24. ***Exercise.*** *Compose a program to calculate the variance of a list of data using the following identity:*

$$s^2 = \frac{n\sum x^2 - (\sum x)^2}{n(n-1)}$$

The symbol $\sum x^2$ means: lift each datum to the second power and add everything. The symbol $(\sum x)^2$ means: add all data and lift to the square.

To make your program into a helping tool for teachers and students, you must have it to output enough intermediate results. In that way, they would compare each important step of their calculations with the output of your program.

25. ***Motherland style.*** *There are infinitely many different styles of programming. Let us rewrite the code developed in the previous exercise in a style according to which one enables reuse but relies on of global variables which assign to memory the role of* ***motherland****. To enable reuse, one encapsulates small pieces of code into separate methods that eventually might serve in others programs. But the output of each method is void and so methods are like agents that go around the whole country making things and eventually cropping outputs. The code follows:*

```
//Program F25 meanVarList3
//This program computes the mean and
//variance of a list of data.
//Motherland style of design:
```

```
//Code is compartmentalized in methods
//but all inputs and outputs of  methods are void
//and so they work over globally defined variables
//that assign to memory the role of motherland
//and to methods that of servicing agents.

public class meanVarList3
{

 /*
  * Variables defined here
  * at the very beginning of the program
  * are defined as static. They
  * remain alive during the whole execution
  * and can be read or updated by every method.
  * They are global variables.
  */

//Example 19 page 9.
private static double  DataVect[]  = {5, 3, 4.5, 2};
private static double sumx=0, sumx2=0;
private static double  mean;
private static int n = DataVect.length;
//Symbolic initialization
private static double  squaresVect[]  = {0,0,0,0};
private static double var,  deviation, coeffVar;

//Sums data in vector with real numbers
private static void sumVector()
 {
  for ( int i = 0; i < n; i++ )
  sumx = sumx + DataVect[i];
 }

//The mean of a list of data
private static void meanVector()
 {
  int n = DataVect.length;
  mean = sumx / n;
 }

//Computes the squares of each entry of a vector
private static void squareVector( )
 {
  for(int i = 0; i < n; i++ )
  squaresVect[i] = DataVect[i]*DataVect[i];
```

```
  }


//Sums data in vector with real numbers
private static void sumVector2()
 {
  for ( int i = 0; i < n; i++ )
 sumx2 = sumx2 +  squaresVect[i];
 }


private static void printTable()
{
 String a = "x";
 String b = "x2 = x^2";
     System.out.printf("%n %10s  %11s", a, b);
     for(int i = 0; i < n; i++ )
      System.out.printf("%n %10.3f   %10.3f ",
       DataVect[i], squaresVect[i] );
  System.out.printf("%n Sum %6.3f   %10.3f ",
                sumx,        sumx2 );
  System.out.print("\n \n");
  System.out.println("n = number of data = " + n);

  System.out.println("\nsumx = " + sumx);
  System.out.println("Mean = sumx / n");
  System.out.println("Mean = " + mean);
  System.out.println("sumx2 = " + sumx2);
  System.out.println("variance = " +
   "(n * sumx2 - sumx*sumx)/ (n*( n-1))");
  var = (n * sumx2 - sumx*sumx)/ (n*( n-1));
  System.out.println( "Variance = "+ var);
  deviation = Math.pow( var, 0.5);
  System.out.println( "Deviation = "+ deviation);
  coeffVar =  deviation/mean;
  System.out.println( "Coefficient of variation " +
                "= deviation/mean");
  System.out.println( "Coefficient of variation = "
                                  + coeffVar);
 }


private static void explanation()
 {
   System.out.println( "\nCoeff around 0.01, " +
                       "homogeneous population." +
```

```
                        "\nCoeff around 0.1, " +
                        " +/- homogeneous population."+
                        "\nCoeff around 0.5, " +
                        "  heterogenous population.");
  }

//The mean and variance of a list of data
private static void meanVarVector()
 {
  sumVector();
  meanVector();
  squareVector();
  sumVector2();
  printTable();
  explanation();
}



public static void main(String[] args)
 {
   System.out.println( "MEAN AND VAR OF A LIST OF DATA");
   meanVarVector();
 }

}
```

26. ***Exercise.*** *Run the program and play with the code.*

27. ***Exercise.*** *Examine the previous code to judge whether or not compartmentalizing code into methods indeed enables reuse, i.e., evolution.*

28. ***Functional style.*** *Let us see a program written in the* ***functional style****. Such a style is distinguished because it enables evolution to the full, where evolution stands for natural and easy reuse of outputs using methods to build more complex objects. To that aim, the first idea is of course to encapsulate functional parts of code into separate methods. Second, to eliminate the need of duplicate code the function paradigm is adopted: every method is a machine that processes in a unique way a given input into a unique output that must be clearly separated from the input. In third place, we declare that global variables are forbidden and instead one creates the necessary objects or classes to encapsulate for reuse the natural output of every method. The immediate advantage of having no global variable is that the complexity of design diminishes. In effect, if global variables are not allowed, every method must define locally the necessary short living helping variables and so it separates domestic problems from national ones*

and also the past from the present. In the next illustrative example we lift the entries of a vector to the second power.

```
//Program F28 functionStyle
//No global variables,
//input and output are clearly separated,
//output is ready for reuse.

public class functionStyle
{

//Prints data in vector
private static void printVector(double[] Vect)
 {
  //System.out.println( "Data are" );
  for ( int i = 0; i < Vect.length; i++ )
   System.out.println( Vect[ i ]  );
 }

//The entries of Vect are squared
private static double[] squareVector(double[] Vect)
 {
   int n = Vect.length;
   //Symbolic initialization
   double  Vect2[]  = {0,0,0,0};
   for ( int i = 0; i < n; i++ )
   Vect2[i] =  Vect[i]*Vect[i];
   return Vect2;
 }


public static void main(String[] args)
 {
   System.out.println( "DATA");
   double  DataVect[]  = {5, 3, 4.5, 2};
   printVector(DataVect);
   double  DataVect2[]  = squareVector(DataVect);
   System.out.println( "SQUARES OF DATA ");
   //The output of squareVector() is used by printVector()
   printVector(DataVect2);
 }

}
```

29. ***Exercise.*** *Run the program and play with the code.*

30. ***Exercise.*** *Rewrite according to the functional style the code developed to calculate the mean and variance of a list of data.*

31. ***Own objects.*** *OOP (object oriented programming) is distinguished because everything is ready for easy reuse. Java is not completely OOP because this language has many purposes, one of which is to allow beginners to have a happy beginning. So, many items actually are not objects. So, when trying to make a connection, it fails. The solution is to devise own objects. The present level of complexity is appropriate to learn/review how this is done. The next code devises an object, Vector, which is just a one dimensional array of type double. The code follows:*

```
//Program F31 meanVarList5
/* This program computes the mean and
   variance of a list of data.
   Functional style of design is implemented
   in the form of  an inner class or object:
   */

public class meanVarList5
{

 /*
  * An object is defined is the form of an inner class
  */


  //The outer application is instantiated
  private static meanVarList5 p = new meanVarList5();


  //======== Inner class definition=======

  //This inner class defines a new type: Vector.
  //It is like types int or double.
  //It converts array F (one dimensional) into
  //a class or object, ready for reuse.
  private  class Vector
  {
//The class Vector fundamentally has two elements:
//a vector and its length.
int length;
double F[] = new double[2000];

//The input of Vector is defined here
//by means of an initialization procedure
```

```
//that is called constructor.
//An instance of Vector can be
//initialized in various ways,
//with various constructors:

//First: Automatic zeroed initialization
Vector(int l )
{
length = l;
for(int i = 0; i < length; i++)
 F[i] = 0;
}

//Second: Initialization from  a 1d- array
Vector(double[] A, int m )
{
 length = m;
 for(int i = 0; i < length; i++)
 F[i] = A[i];
}

//Third: Initialization by cloning from a Vector
Vector(Vector A)
{
 length = A.length;
 for(int i = 0; i < A.length; i++)
 F[i] = A.F[i];
}


//In the same way as 5 is an instance of int,
//so, one can construct specific instances of Vector.
//They can be manipulated as shown below.
//But they are also allowed    to produce outputs
//or actions on their own such as this:
private  void printVector(Vector Vect, String s)
{
 System.out.println( "\n" + s );
 for ( int i = 0; i < Vect.length; i++ )
  System.out.println( Vect.F[ i ]  );
 System.out.println( );
}
}//end of inner class

//=== Methods of the outer class meanVarList5 ===
```

```
//Data are summed.
//The input is a Vector, the output is of type double.
private static double sumVector(Vector V)
 {
  double sum = 0;
  for(int i = 0; i < V.length; i++)
sum  = sum +  V.F[i];
  return sum;
 }


//The mean of a list of data
private static double meanVector(Vector V)
 {
  int n = V.length;
  double mean =  sumVector(V) / n;
  return mean;
 }


//Every entry of a Vector is squared.
//The input is a Vector, the output is also a Vector.
private static Vector squareVector(Vector V)
{
 //Zeroed initialization of a Vector
 Vector  list = p.new Vector(V.length);
 //The square of each entry of F is calculated
 //and kept in list.F
 for(int i = 0; i < V.length; i++)
list.F[i] = V.F[i] * V.F[i];
 return list;
}

//Documentation is printed
private static void explanation()
{
  System.out.println( "\nCoeff around 0.01, " +
                      "homogeneous population." +
                      "\nCoeff around 0.1, " +
                      " +/- homogeneous population."+
                      "\nCoeff around 0.5, " +
                      "  heterogenous population.");
 }


//This is the core of the main class
```

```
//Here, commands are given to execute the overall task.
private static void meanVarVector(Vector V)
{
 String a = "x";
 String b = "x2 = x^2";
 System.out.printf("%n %10s  %11s", a, b);
 int n = V.length;
 Vector squareVect = p.new Vector(V);
 squareVect = squareVector(V);
 for(int i = 0; i < n; i++ )
    System.out.printf("%n %10.3f   %10.3f ",
       V.F[i], squareVect.F[i] );
 double sumx = sumVector(V);
 double mean = meanVector(V);
 double sumx2 = sumVector(squareVect);
 System.out.printf("%n Sum %6.3f   %10.3f ",
                sumx,        sumx2 );
 System.out.print("\n \n");
 System.out.println("n = number of data = " + n);

 System.out.println("\nsumx = " + sumx);
 System.out.println("Mean = sumx / n");
 System.out.println("Mean = " + mean);
 System.out.println("sumx2 = " + sumx2);
 System.out.println("variance = " +
   "(n * sumx2 - sumx*sumx)/ (n*( n-1))");
 double var = (n * sumx2 - sumx*sumx)/ (n*( n-1));
 System.out.println( "Variance = "+ var);
 double deviation = Math.pow( var, 0.5);
 System.out.println( "Deviation = "+ deviation);
 double coeffVar =  deviation/mean;
 System.out.println( "Coefficient of variation " +
                 "= deviation/mean");
 System.out.println( "Coefficient of variation = "
                                  + coeffVar);
 explanation();
}

//Input is read and directed to processing methods
public static void main(String[] args)
{
  System.out.println( "MEAN AND VAR OF A LIST OF DATA");
  //Example 19 page 9.
  double  DataVect[]  = {5, 3, 4.5, 2};
  //Initialization of a Vector from an array
  Vector  list1 = p.new Vector(DataVect, DataVect.length);
```

```
  list1.printVector(list1, "Original data");
  meanVarVector(list1);
}


}//End of main classes
```

32. ***Exercise.*** *Run the program and play with the code. To better understand the code, switch your Eclipse to debugging mode as follow:*

1. In the editor window, in the most left column, right-click in front of the instruction that you want to understand and choose Toggle Breakpoint. A blue circle will appear in the left column.

2. In the main menu, choose: Run → Debug As → Java Application

3. Make sure that the console is visible: in the main menu, choose window → view → Console.

3. Press F5 various times and observe that the output appears step by step.

4. Observe that F5 is one among various possibilities of 'debugging' this program. Search out for others modes under the run menu and play with them.

5. To return to ordinary mode, go to Window → Open Perspective → Java.

33. ***Gossipless, transparent style.*** *There are infinitely many styles. Say, one can be eventually very proud of the previous program that follows the functional style and that moreover devises own objects. Nevertheless, we have at once a criticism: the problem spins around a table that contains data and their squares. Nevertheless, such a table appears as a visual illusion in the console once it is printed but the program contains no table in its data structure or in its action flow. Therefore, that program can be blamed for its lack of transparency. So, a* ***transparent style*** *is one in which the code, its output, the problem and its solution are all made on the same image. The importance of transparency stems form the fact that developing software is just going over fixing bugs whose correction generates more bugs. So, transparency is important because it enables easier correction of bugs.*

As everyone knows, gossips add to life a lot of troubles. The same happens in programming. Thus, we say that a ***style is gossipless*** *when domestic problems are solved at home while problem found at the place of work are solve precisely there. The form as this directive is implemented is by defining local variables that do not exist outside the scope of local problems. This style has been adopted by Java.*

34. ***The code for a transparent version of the program that calculates a variance follows.***

```
//Program F34 meanVarList6
/* This program computes the mean and
   variance of a list of data.
   Transparent, functional style of design:
```

```
   1) Code is compartmentalized in methods.
   2) Every method is a function with
      a clearly specified input and
      with an output that is ready for reuse
      in other methods.
   3) No global variables.
   4) Methods  are ready for reuse in other classes.
   5) The code, its output, the problem and its solution
      are all made on the same image.
   */

public class meanVarList6
{

 /*
  * No global variables
  */


 //Data of column k are summed.
 //The input is a table, the output is of type double.
 private static double sum(double[][]  T, int k)
 {
  double sum = 0;
  for(int i = 0; i < T.length; i++)
sum  = sum +  T[i][k];
  return sum;
 }

//A matrix is constructed with two columns.
//First: raw data, second, teir squaresl.
private static double[][] buildTable(double[] V)
{
  //Zeroed initialization of a Vector
  double[][]  M = new double[V.length][2];
  //The square of each entry of F is calculated
  //and kept in list.F
  for(int i = 0; i < V.length; i++)
{
 M[i][0] = V[i];
 M[i][1] = V[i] * V[i];
}
return M;
}

//A table is printed
```

```
private static void printTable(double[][] T)
{
 String a = "x";
 String b = "x2 = x^2";
 System.out.printf("%n %10s  %11s", a, b);
 int n = T.length;
 for(int i = 0; i < n; i++ )
    System.out.printf("%n %10.3f   %10.3f ",
       T[i][0], T[i][1] );
}

//Documentation is printed
private static void explanation()
{
  System.out.println( "\nCoeff around 0.01, " +
                      "homogeneous population." +
                      "\nCoeff around 0.1, " +
                      " +/- homogeneous population."+
                      "\nCoeff around 0.5, " +
                      "  heterogenous population.");
 }


//This is the core of the main class
//Here, commands are given to execute the overall task.
private static void meanVarVector(double[] V)
{
 //A table is declared
 double[][]  Table = new double[V.length][2];
 Table = buildTable(V);
 printTable(Table);
 double sumx = sum(Table,0);
 int n = Table.length;
 double mean = sumx/ n;
 double sumx2 = sum(Table,1);
 System.out.printf("%n Sum %6.3f   %10.3f ",
                sumx,        sumx2 );
 System.out.print("\n \n");
 System.out.println("n = number of data = " + n);

 System.out.println("\nsumx = " + sumx);
 System.out.println("Mean = sumx / n");
 System.out.println("Mean = " + mean);
 System.out.println("sumx2 = " + sumx2);
 System.out.println("variance = " +
   "(n * sumx2 - sumx*sumx)/ (n*( n-1))");
```

```
 double var = (n * sumx2 - sumx*sumx)/ (n*( n-1));
 System.out.println( "Variance = "+ var);
 double deviation = Math.pow( var, 0.5);
 System.out.println( "Deviation = "+ deviation);
 double coeffVar =  deviation/mean;
 System.out.println( "Coefficient of variation " +
                "= deviation/mean");
 System.out.println( "Coefficient of variation = "
                                + coeffVar);
 explanation();
}

//Input is read and directed to processing methods
public static void main(String[] args)
{
  System.out.println( "MEAN AND VAR OF A LIST OF DATA");
  //Example 19 page 9.
  double  DataVect[]  = {5, 3, 4.5, 2};
  meanVarVector(DataVect);
}

}
```

35. ***Exercise.*** *Run the program and play with the code.*

36. ***Panacea style.*** *We tacitly are engaged in the task of discovering a style of design that will help us in the very difficult task of software development. So, it seems that we are looking for a panacea. Are we correctly oriented?*

37. ***Exercise.*** *Develop a program in which a table becomes an object in the same way as a vector was made in a personalized object. Make your best to follow a gossipless, transparent, functional style of design.*

2.2 Grouping

To present a list of data is not the best option for all purposes. Because of various reasons one would prefer to group data in classes, ranges or intervals. The first case is considered in frequencies tables that by default report absolute frequencies of repeated data.

38. ***Frequency tables.*** *If data are repeated, as in the scores of a group of students, one might group data in a* ***frequency table****, such that datum x is repeated F times.*

In that case, the mean and variance take the following form.

The mean is

$$\bar{X} = \frac{\Sigma(xF)}{n} = \frac{\Sigma(x_i F_i)}{n} = \frac{(x_1 F_1)+....+(x_n F_m)}{n}$$

The variance can be calculated variously, we choose one method that is suitable for work with calculators. It is the output of the following procedure:

Step 1: the pondered sum of squares is calculated

$$\Sigma x^2 F = x_1^2 F_1 + x_2^2 F_2 + ... + x_n^2 F_m$$

Step 2: the sum of squares S_{xx} term is calculated

$$S_{xx} = \Sigma x^2 F - \frac{(\Sigma xF)^2}{n}$$

Step 3: the variance is:

$$s^2 = \frac{S_{xx}}{n-1}$$

39. *Exercise.* *Compose an OOP program to calculate mean and variance of a frequency table. It is by no mean sure that devising own objects might be the best option for this problem, but we use the occasion to practice with OOP.*

40. *Evolutionary inertia.* *From the stand point of reuse, evolution might become a heavy load for development. To see this, let us consider that the computing of the variance can be implemented in at least two ways. If we want to reuse the code that was developed to calculate the variance of a list of data, we would devise a table with five columns: x, F, xF, x^2, x^2F. In that way, we use the code to calculate squares and we would understand x^2F as $(x^2)F$. Nevertheless, if we do not want to rely on reuse but on the best option, we might use only four columns: x, F, xF, x^2F and we would understand x^2F as $x(xF)$. We cannot exclude that in a good evolutionary environment both alternatives would appear and that the best would win in the long trend. But the mater is that evolution must left tracks in the fossil record in which non optimal solutions must be found all around the world. In our case, this prediction is correct: one can see that the optimal solution can be found in version 7 of the first vol of Statistics that accompanies this series but the suboptimal solution can be found in all previous editions. This fact illustrates our general claim: evolution of complex features must left a clear evolution towards perfection that must be apparent in the fossil record.*

41. *Exercise.* *Adapt the code that was developed for a frequency table to find the mean and variance of a cumulative frequency table. In such a table, the frequency associated to each class marker includes as its own frequency as all those below it.*

42. *From a list to a frequency table.* *If we want to extract the tendency of a given set of data, say, to determine it they follow a central one, one must calculate a table of absolute frequencies that next could be represented by, say, a bar diagram. To pass from the list to a frequency table, on must group data by intervals. This can be done in one of two forms. First, to explicitly define the borders of each class or interval and second, to define the first inferior limit and the length of each interval. The code for the first method follows:*

```
//Program F42 ListToFT
//Groups data in interval classes.
//Input:
//A list of data,
//a list of the borders of the intervals,
//and a  list of class markers.
//Outputs a frequency table with its
//mean and variance.
//Style reuse.

public class ListToFT
{

 //The outer application is instantiated
 private static ListToFT p = new ListToFT();


 //======== Inner class definition=======

 //This inner class defines a new type: Table.
 //It converts array F (two dimensional) into
 //a class or object, ready for reuse.
 private  class Table
  {
   //The class Table has two main elements:
   //1) The titles of  the columns, whose length
   //gives the number of columns.
   //2) A matrix, whose length is the number of rows.

   String[] Titles = { "x",  "F",   "xF", "x2F"};
   double M[][] = new double[2000][Titles.length];
   int nRows;
   //The input of Table is defined here
   //by means of an initialization procedure
   //that is called constructor.
   //The natural input is the frequency table
   //which is consigned in a matrix FT with 2 columns.
 Table(double[][] FT )
  {
   nRows = FT.length;
   //The matrix M of Table is initialized
   //Raw data go in   columns 0 and 1.
   //Squares in the second.
   for(int i = 0; i < nRows; i++)
    {
 M[i][0] = FT[i][0];
```

```
 M[i][1] = FT[i][1];
 M[i][2] = M[i][0] * M[i][1];
 M[i][3] = M[i][0] * M[i][2];
}
   }

 //Data of column k are summed.
 //The input is a table, the output is of type double.
 private  double sum(Table  T, int k)
  {
   double sum = 0;
   for(int i = 0; i < T.nRows; i++)
  sum  = sum +  T.M[i][k];
   return sum;
  }

 private  void printTable(Table T)
  {
   System.out.printf("%n %10s %10s %10s %10s",
             T.Titles [0], T.Titles [1],
              T.Titles [2],T.Titles [3]);
   for ( int i = 0; i < T.nRows; i++ )
    System.out.printf(
"%n %10.3f %10.3f %10.3f %10.3f ",
T.M[i][0], T.M[i][1], T.M[i][2], T.M[i][3] );
System.out.println( );
   }
  }//end of inner class



//=== Methods of the outer class ListToFT  ===

 private static void printList(double DataVector[],
                               String s)
 {
System.out.println( "\n " +s );
int nData = DataVector.length;

boolean go = true;
int nCols = 10;
int nRows = 9;
int counter = 0;
   for (int i = 0; (i < nRows) & go; i++)
    {
  for (int j = 0; (j < nCols) & go; j++)
```

```
  {
  System.out.print( DataVector[i*10 + j] + "   ");

  if (counter >= nData-1) go = false;
  counter = counter+1;
  }
  System.out.println( );
 }
System.out.print("\nNumber of data = " +nData + "\n");

 }

//List is the input, a frequency table is the output
private static double[][] group(double List[], double Borders[],
                          double classMarkers[]  )
{
 int nClasses = classMarkers.length;
 double FreqTable[][] = new double[nClasses][2];
 int nData = List.length;

 for(int j=0; j < nClasses; j++)
 {
 FreqTable[j][0] = classMarkers[j];
 FreqTable[j][1] = 0;
 }
 for(int i=0; i< nData; i++)
 {
for(int j=0; j < nClasses; j++)
{
  if ((Borders[j] <= List[i]) & (List[i] < Borders[j+1]))
  FreqTable[j][1] =FreqTable[j][1] +1;
}

 }

 return FreqTable;
}


//Documentation is printed
private static void explanation()
 {
  System.out.println( "\nCoeff around 0.01, " +
                        "homogeneous population." +
                        "\nCoeff around 0.1, " +
                        " +/- homogeneous population."+
```

```
                        "\nCoeff around 0.5, " +
                        "  heterogenous population.");
 }


// Mean and variance of a frequency table
private static void meanVarTable(double FreqTable[][])
{
 //A table is declared
 Table T =  p.new   Table(FreqTable);
 T.printTable(T);
 double n = T.sum(T,1);
 double sumxF = T.sum(T,2);
 double sumx2F = T.sum(T,3);
 System.out.printf("Sum %18.3f %10.3f %10.3f ",
             n,   sumxF,        sumx2F );
 double mean = sumxF/ n;
 System.out.print(
  "\n\n n = Number of  objects = sum F = "+ (int) n);
 System.out.printf( "\n sum xF  = %10.3f ", sumxF);
 System.out.printf( "\n sum x2F  = %10.3f ",  sumx2F);
 System.out.printf( "\n Mean = sum xF / n = %10.3f ",
             mean);
 double Sxx  = sumx2F -    Math.pow(sumxF, 2)/n;
 System.out.printf(
 "\n Sxx = sum x2F - (sumxF)^2 /n = %10.3f ", Sxx);
 double var = Sxx / (n-1);
 System.out.printf(
 "\n Variance =  Sxx/(n-1) = %10.3f ", var);
 double deviation = Math.sqrt(var);
 System.out.printf(
 "\n Deviation =  %10.3f ", deviation);
 double c = deviation/ mean;
 System.out.printf(
    "\n Coefficient of variation =  %10.3f \n", c);
 explanation();
}


public static void main(String[] args)
{
//Declaration of data:
//observed values as a list
//Example 24 pag 12
double    List[] =
{
```

```
48.5, 49.2, 51.0, 50.9, 48.7, 50.5, 49.5, 50.8, 50.0, 51.1,
50.7, 51.1, 49.3, 49.1, 50.0, 48.7, 51.2, 49.2, 49.0, 49.3,
49.1, 48.7, 48.6, 49.2, 49.9, 50.1, 50.1, 50.9, 52.4, 50.2,
49.5, 50.9, 51.4, 49.7, 49.8, 50.8, 50.3, 51.8, 50.0, 51.1,
51.3, 50.1, 50.4, 51.1, 49.8, 49.8, 50.1, 50.7, 50.1, 49.9,
49.7, 51.1, 49.7, 49.9, 49.8, 50.5, 49.7, 50.8, 50.7, 50.7,
50.8,  51.5, 51.0
};
System.out.println("FROM DATA LIST TO FREQUENCY TABLE:");
printList(List, "Raw data");
//Borders of the intervals
double  Borders[] = {48.5, 49.5, 50.5, 51.5, 52.5};
double classMarkers[] = {49, 50, 51, 52};
int nClasses = classMarkers.length;
double FreqTable[][] = new double[nClasses][2];
printList(Borders, "Borders of intervals");
FreqTable = group(List, Borders, classMarkers );
meanVarTable(FreqTable);
}

}//end of class
```

43. ***Exercise.*** *Run the program and play with the code. Explain why the style of this program is reuse but not transparent. Discuss the implications over the dilemma of evolution vs good style.*

44. ***Exercise: from a data list to a frequency table, variant.*** *Reuse previous codes to devise a program that computes a frequency table from a data list given the first border of the inferior interval and the length of intervals. Mean and variance must be reported.*

2.3 Toolboxes

A **toolbox** is a program whose main method is void so that it can be filled at pleasure by the User by making calls to needed methods. In practice, one fills the main method with a list of all available ones but they are muted with the possible exception of one, which is active.

45. ***Example.*** *A toolbox looks like this:*

```
//Program E45 toolBox1
//This is how a
//toolbox looks like.

/*
 This is a tool box of methods to calculate some statistics
```

```
 of a list of data and of frequency tables.

 How to use this program:

 1. This program is a tool box because its main method
    contains  all relevant allowed methods
    but only one is activated, all others are
    muted.
 2. To silence one method, append // as a prefix.
    To activate a method, delete the prefix //.
 3. For a given task, choose the best suited method
    among the diverse variants.
 4. To run a method with your own data,
    modify the inbuilt data according to your needs.
 5. To inquire about unknown terms, drag the mouse over
    the term and wait until Eclipse displays the
    corresponding documentation.
 6. To go   from the name of a
    method  to its implementation:
    Pose the cursor over the name of the method
    and press  F3: you will be transported to the
    implementation of the method.
 7. To mark a specific site in the program:
     -Activate line numbering.
     -Pose the cursor over the line you want to mark.
     -Go to the corresponding number at the left margin
      of the Editor Window and right click.
     -Choose the Add Bookmark menu and punch it.
     -A blue marker will appear in the right margin of the
      Editor Window. By clicking over it, one can
      return to the desired place.
      One can also play with the menu
      Window -> ShowView -> Outline:
      a list with all methods will appear,
      click over any one and you will
      transported to the corresponding place
      along the text of the program.

*/

public class toolBox1
{

//Processes a string
private static void method1(String s)
 {
```

```
   System.out.println(s);
 }

//Processes a string and an int
private static void method2(String s, int n)
 {
   int t = s.length()*n;
   System.out.println("s = " + s);
   System.out.println("n = " + n);
   System.out.println("t = s.length()*n");
   System.out.println("t = " + t);
 }

//Processes a vector
 private static void method3(double Vect[])
 {
   double sum =  0;
   for(int i = 0; i < Vect.length; i++)
    sum = sum + Vect[i];
   System.out.println("Sum = " + sum);
 }


 public static void main(String[] args)
 {
  //method1("abc");
  //method2("Hello!",4);
  double Data[] = {1,2,3,4,5};
  method3(Data);
 }

}//End of class.
```

46. ***Exercise.*** *Run the program and give each method the possibility to get activated while the others remain silent.*

47. ***Example: a toolbox with just one tool.*** *Let us make a toolbox to calculate the basic statistics of a list of data and nothing else. We look for the best of available options and include it in our toolbox.*

```
//Program F47 toolBox2
/*
 This is a tool box of methods to calculate
 some statistics of a list of data.
```

```
 How to use this program:

 1. Append it to any project and run it.
 2. This program is a tool box because its main method
    can be filled with any allowed method.
 3. Most relevant allowed methods are listed in the main
    but only one is activated, all others are
    muted.
 4. To silence one method, append // as a prefix.
    To activate a method, delete the prefix //.
 5. For a given task, choose the best suited method
    among the diverse variants.
 6. To run a method with your own data,
    modify the inbuilt data according to your needs.
 7. This first version has just one aim:
    To implement fundamental tasks as transparently
    as possible. So, it looks rather like an array of
    independent, non interacting  methods.
 8. To inquire about unknown terms, drag the mouse over
    the term and wait until Eclipse displays the
    corresponding documentation.
 9. To go   from the name of a
    method  to its implementation:
     Pose the cursor at the end of the line
     with the name of the method and press the space bar,
     next press F3: you will be transported to the
     implementation of the method.
 10. To mark a specific site in the program:
     -Activate line numbering.
     -Pose the cursor over the line you want to mark.
     -Go to the corresponding number at the left margin
      of the Editor Window and right click.
     -Choose the Add Bookmark menu and punch it.
     -A blue marker will appear in the right margin of the
      Editor Window. By clicking over it, one can
      return to the desired place.
      One can also play with the menu
      Window -> ShowView -> Outline
      A list with all methods will appear:
      click over any one and you will
      transported to the corresponding place
      along the text of the program.

*/
```

```
public class toolBox2
{

 //The outer application is instantiated
 private static toolBox2 p = new toolBox2();


//============================
//MAIN--MAIN--MAIN--MAIN--MAIN
//============================

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


//=============================================
//          IMPLEMENTATION
//=============================================


 //======== Inner class definition=======

 //This inner class defines a new type: Table.
 //It converts array F (two dimensional) into
 //a class or object, ready for reuse.
 private  class Table
 {
  //The class Table fundamentally has three elements:
  //a matrix with two columns and its length,
  //which is the number of rows and
  //the titles of columns.
  int length;
  String[] Titles = { "x",  "x2 = x^2"};
  double M[][] = new double[2000][2];

  //The input of Table is defined here
  //by means of an initialization procedure
  //that is called constructor.
  //The natural input is the raw data,
  //which is consigned in a one dimensional array.
  Table(double[] V )
  {
   length = V.length;
   //The matrix M of Table is initialized
```

```
   //Raw data go in the first column.
   //Squares in the second.
   for(int i = 0; i < length; i++)
   {
 M[i][0] = V[i];
 M[i][1] = V[i]*V[i];
   }
  }

  //Data of column k are summed.
  //The input is a table, the output is of type double.
  private  double sum(Table  T, int k)
  {
   double sum = 0;
   for(int i = 0; i < T.length; i++)
  sum  = sum +  T.M[i][k];
   return sum;
  }

 private  void printTable(Table T)
  {
    System.out.printf("%n %10s  %11s", T.Titles [0], T.Titles [1]);
 for ( int i = 0; i < T.length; i++ )
    System.out.printf("%n %10.3f   %10.3f ",
        T.M[i][0], T.M[i][1] );
 System.out.println( );
  }
}//end of inner class


//=== Methods of the outer class toolBox2 ===


//Prints the list and the title
private static void printList(double DataVector[],
                              String title)
{
 System.out.println( "\n " + title );
 int nData = DataVector.length;
 boolean go = true;
 int nCols = 10;
 int nRows = 9;
 int counter = 0;
 for (int i = 0; (i < nRows) & go; i++)
  {
   for (int j = 0; (j < nCols) & go; j++)
```

```
    {
     System.out.print( DataVector[i*10 + j] + "   ");
     if (counter >= nData-1) go = false;
     counter = counter+1;
    }
  System.out.println( );
 }
 System.out.print("\nNumber of data = " +nData + "\n");
}

//Documentation is printed
private static void explanation()
{
  System.out.println( "\nCoeff around 0.01, " +
                      "homogeneous population." +
                      "\nCoeff around 0.1, " +
                      " +/- homogeneous population."+
                      "\nCoeff around 0.5, " +
                      "  heterogenous population.");
 }


//This is the core of the main class
//Here, commands are given to execute the overall task.
private static void meanVarVector(double[] V)
{
 //A table is declared
 Table T =  p.new   Table(V);
 T.printTable(T);
 double sumx = T.sum(T,0);
 int n = T.length;
 double mean = sumx/ n;
 double sumx2 = T.sum(T,1);
 System.out.printf("%n Sum %6.3f   %10.3f ",
                sumx,        sumx2 );
 System.out.print("\n \n");
 System.out.println("n = number of data = " + n);

 System.out.println("\nsumx = " + sumx);
 System.out.println("Mean = sumx / n");
 System.out.println("Mean = " + mean);
 System.out.println("sumx2 = " + sumx2);
 System.out.println("variance = " +
   "(n * sumx2 - sumx*sumx)/ (n*( n-1))");
 double var = (n * sumx2 - sumx*sumx)/ (n*( n-1));
 System.out.println( "Variance = "+ var);
```

```
 double deviation = Math.pow( var, 0.5);
 System.out.println( "Deviation = "+ deviation);
 double coeffVar =  deviation/mean;
 System.out.println( "Coefficient of variation " +
                 "= deviation/mean");
 System.out.println( "Coefficient of variation = "
                                  + coeffVar);
 explanation();
}

//Input is read and directed to processing methods
public static void meanVarList( )
{
  System.out.println( "MEAN AND VAR OF A LIST OF DATA");
  //Example 19 page 9.
  double  List[]  = {5, 3, 4.5, 2};
  //Prints list and title
  printList(List,  "Raw data");
  System.out.println( "\nTABLE WITH x AND ITS SQUARE");
  meanVarVector(List);
}

}
```

48. ***Exercise.*** *Run the program and play with the code.*

49. ***Example: how to add a new function to the toolbox.*** *Let us add now to our toolbox the possibility to calculate the mean and variance of a frequency table. Given that we have got content with only one test for each program, we cannot guarantee that our programs are bugfree. Therefore, the only aim of the toolbox must be at present just to function. No other improvement is envisaged, in particular simplification is forbidden if it hinders the easiness to debug or modify the material.*

```
//Program F49 toolBox3
/*
 This is a tool box of methods to calculate
 some statistics of a list of data.

 How to use this program:

 1. Append it to any project and run it.
 2. This program is a tool box because its main method
    can be filled with any allowed method.
 3. Most relevant allowed methods are listed in the main
    but only one is activated, all others are
    muted.
```

```
 4. To silence one method, append // as a prefix.
    To activate a method, delete the prefix //.
 5. For a given task, choose the best suited method
    among the diverse variants.
 6. To run a method with your own data,
    modify the inbuilt data according to your needs.
 7. This first version has just one aim:
    To implement fundamental tasks as transparently
    as possible. So, it looks rather like an array of
    independent, non interacting  methods.
 8. To inquire about unknown terms, drag the mouse over
    the term and wait until Eclipse displays the
    corresponding documentation.
 9. To go   from the name of a
    method  to its implementation:
     Pose the cursor at the end of the line
     with the name of the method and press the space bar,
     next press F3: you will be transported to the
     implementation of the method.
10. To mark a specific site in the program:
     -Activate line numbering.
     -Pose the cursor over the line you want to mark.
     -Go to the corresponding number at the left margin
      of the Editor Window and right click.
     -Choose the Add Bookmark menu and punch it.
     -A blue marker will appear in the right margin of the
      Editor Window. By clicking over it, one can
      return to the desired place.
      One can also play with the menu
      Window -> ShowView -> Outline
      A list with all methods will appear:
      click over any one and you will
      transported to the corresponding place
      along the text of the program.

*/


public class toolBox3
{

 //The outer application is instantiated
 private static toolBox3 p = new toolBox3();
```

```
//============================
//MAIN--MAIN--MAIN--MAIN--MAIN
//============================

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



//============================================
//               IMPLEMENTATIONS
//============================================



//============================================
//       MEAN AND VAR O A LIST OF DATA
//============================================



 //======== Inner class definition=======

 //This inner class defines a new type: Table.
 //It converts array F (two dimensional) into
 //a class or object, ready for reuse.
 private  class Table
 {
  //The class Table fundamentally has three elements:
  //a matrix with two columns and its length,
  //which is the number of rows and
  //the titles of columns.
  int length;
  String[] Titles = { "x",  "x2 = x^2"};
  double M[][] = new double[2000][2];

  //The input of Table is defined here
  //by means of an initialization procedure
  //that is called constructor.
  //The natural input is the raw data,
  //which is consigned in a one dimensional array.
  Table(double[] V )
```

```
  {
   length = V.length;
   //The matrix M of Table is initialized
   //Raw data go in the first column.
   //Squares in the second.
   for(int i = 0; i < length; i++)
   {
 M[i][0] = V[i];
 M[i][1] = V[i]*V[i];
   }
  }

  //Data of column k are summed.
  //The input is a table, the output is of type double.
  private  double sum(Table  T, int k)
  {
   double sum = 0;
   for(int i = 0; i < T.length; i++)
  sum  = sum +  T.M[i][k];
   return sum;
  }

 private  void printTable(Table T)
  {

    System.out.printf("%n %10s  %11s", T.Titles [0], T.Titles [1]);
 for ( int i = 0; i < T.length; i++ )
    System.out.printf("%n %10.3f   %10.3f ",
        T.M[i][0], T.M[i][1] );
 System.out.println( );
  }
}//end of inner class


//=== Methods of the outer class toolBox3 ===


//Prints the list and the title
private static void printList(double DataVector[],
                              String title)
{
 System.out.println( "\n " + title );
 int nData = DataVector.length;
 boolean go = true;
 int nCols = 10;
 int nRows = 9;
```

```
 int counter = 0;
 for (int i = 0; (i < nRows) & go; i++)
  {
   for (int j = 0; (j < nCols) & go; j++)
    {
     System.out.print( DataVector[i*10 + j] + "   ");
     if (counter >= nData-1) go = false;
     counter = counter+1;
    }
  System.out.println( );
 }
 System.out.print("\nNumber of data = " +nData + "\n");
}

//Documentation is printed
private static void explanation()
{
  System.out.println( "\nCoeff around 0.01, " +
                      "homogeneous population." +
                      "\nCoeff around 0.1, " +
                      " +/- homogeneous population."+
                      "\nCoeff around 0.5, " +
                      "  heterogenous population.");
 }


//This is the core of the main class
//Here, commands are given to execute the overall task.
private static void meanVarVector(double[] V)
{
 //A table is declared
 Table T =  p.new   Table(V);
 T.printTable(T);
 double sumx = T.sum(T,0);
 int n = T.length;
 double mean = sumx/ n;
 double sumx2 = T.sum(T,1);
 System.out.printf("%n Sum %6.3f   %10.3f ",
                sumx,        sumx2 );
 System.out.print("\n \n");
 System.out.println("n = number of data = " + n);

 System.out.println("\nsumx = " + sumx);
 System.out.println("Mean = sumx / n");
 System.out.println("Mean = " + mean);
 System.out.println("sumx2 = " + sumx2);
```

```
 System.out.println("variance = " +
   "(n * sumx2 - sumx*sumx)/ (n*( n-1))");
 double var = (n * sumx2 - sumx*sumx)/ (n*( n-1));
 System.out.println( "Variance = "+ var);
 double deviation = Math.pow( var, 0.5);
 System.out.println( "Deviation = "+ deviation);
 double coeffVar =  deviation/mean;
 System.out.println( "Coefficient of variation " +
                 "= deviation/mean");
 System.out.println( "Coefficient of variation = "
                                  + coeffVar);
 explanation();
}

//Input is read and directed to processing methods
private static void meanVarList( )
{
  System.out.println( "MEAN AND VAR OF A LIST OF DATA");
  //Example 19 page 9.
  double  List[]  = {5, 3, 4.5, 2};
  //Prints list and title
  printList(List,  "Raw data");
  System.out.println( "\nTABLE WITH x AND ITS SQUARE");
  meanVarVector(List);
}

//=================================================
// MEAN AND VAR OF A TABLE OF ABSOLUTE FREQUENCIES
//=================================================


//======== Inner class definition=======

//This inner class defines a new type: Table.
//It converts array F (two dimensional) into
//a class or object, ready for reuse.
private  class TableFreq
 {
//The class Table has two main elements:
//1) The titles of  the columns, whose length
//gives the number of columns.
//2) A matrix, whose length is the number of rows.

String[] Titles = { "x",  "F",   "xF", "x2F"};
double M[][] = new double[2000][Titles.length];
int nRows;
```

```
//The input of Table is defined here
//by means of an initialization procedure
//that is called constructor.
//The natural input is the frequency table
//which is consigned in a matrix FT with 2 columns.
TableFreq(double[][] FT )
 {
   nRows = FT.length;
   //The matrix M of Table is initialized
   //Raw data go in   columns 0 and 1.
   //Squares in the second.
   for(int i = 0; i < nRows; i++)
   {
 M[i][0] = FT[i][0];
 M[i][1] = FT[i][1];
 M[i][2] = M[i][0] * M[i][1];
 M[i][3] = M[i][0] * M[i][2];
   }
  }

//Data of column k are summed.
//The input is a table, the output is of type double.
private  double sum(TableFreq  T, int k)
 {
  double sum = 0;
  for(int i = 0; i < T.nRows; i++)
  sum  = sum +  T.M[i][k];
  return sum;
 }

private  void printTable(TableFreq T)
 {
  System.out.println("\nExtended matrix to " +
                "calculate variance:");
  System.out.printf("%n %10s %10s %10s %10s",
            T.Titles [0], T.Titles [1],
             T.Titles [2],T.Titles [3]);
    for ( int i = 0; i < T.nRows; i++ )
 System.out.printf(
"%n %10.3f %10.3f %10.3f %10.3f ",
T.M[i][0], T.M[i][1], T.M[i][2], T.M[i][3] );
  System.out.println( );
 }
}//end of inner class
```

```
//=== Methods of the outer class toolBox3  ===



// Mean and variance of a frequency table
private static void meanVarTable(double FreqTable[][])
{
//A table is declared
TableFreq T =  p.new   TableFreq(FreqTable);
T.printTable(T);
double n = T.sum(T,1);
double sumxF = T.sum(T,2);
double sumx2F = T.sum(T,3);
System.out.printf("Sum %18.3f %10.3f %10.3f ",
             n,   sumxF,        sumx2F );
double mean = sumxF/ n;
System.out.print(
  "\n\n n = Number of  objects = sum F = "+ (int) n);
System.out.printf( "\n sum xF  = %10.3f ", sumxF);
System.out.printf( "\n sum x2F  = %10.3f ",  sumx2F);
System.out.printf( "\n Mean = sum xF / n = %10.3f ", mean);
double Sxx  = sumx2F -    Math.pow(sumxF, 2)/n;
System.out.printf(
 "\n Sxx = sum x2F - (sumxF)^2 /n = %10.3f ", Sxx);
double var = Sxx / (n-1);
System.out.printf(
 "\n Variance =  Sxx/(n-1) = %10.3f ", var);
double deviation = Math.sqrt(var);
System.out.printf(
 "\n Deviation =  %10.3f ", deviation);
double c = deviation/ mean;
System.out.printf(
   "\n Coefficient of variation =  %10.3f \n", c);
explanation();
}

//Prints   table
private static void printMatrix(double Matrix[][],
                        String title)
{
System.out.println( title );
int m = Matrix.length;
 for ( int i = 0; i < m; i++ )
 System.out.println( Matrix[ i ][0]+
        "\t "+ Matrix[ i ][1]);
}
```

```
private static void meanVarFreqTable()
{

System.out.println("\n\nMEAN-VAR OF A FREQUENCY TABLE");
//Frequency table: first coordinate = class marker
//second coordinate = frequency

//Example 20, pag 10
double  FreqT[][] =
        {{4, 3},
         {7, 6},
         {10, 10},
         {13, 15},
         {16, 4},
         {19, 2}};

printMatrix(FreqT, "\n Raw data");
meanVarTable(FreqT);
}

}
```

50. ***Exercise.*** *Run the program and play with the code.*

51. ***Exercise.*** *Retro-engineer the previous program toolBox3: merge toolBox2 with the program to calculate mean and var of a frequency table into a single program and make the necessary amendments to achieve a functional unit.*

52. ***Exercise.*** *Embed in your toolBox all the functions implemented in the present chapter to calculate the basic statistics of quantitative data.*

53. ***Editor code for PsTricks.*** *We can use Java to produce the code to draw a bar diagram for PsTricks, a drawing facility for LaTeX. The code follows:*

```
//Code F53
//This is not a class, just code to insert somewhere.
//Produces the pstricks  code for the bar graph
//of a frequency table
public static void makeCode(double FreqTable[][])
{
 System.out.println("\nPSTRICKS code for the FreqTable\n");
 int n = FreqTable.length;
 //Width of each column
 double deltaX = 4;
 //The minimum x-coordinate is found
 double Minxx = 1000;
```

```
 for ( int i = 0; i < n; i++ )
  if (FreqTable[i][0] < Minxx)
        Minxx =   FreqTable[i][0];
 int Minxxx = (int) (deltaX*Minxx - deltaX);
 //The maximum x-coordinate is found
 double Maxx = 0;
 for ( int i = 0; i < n; i++ )
  if ( FreqTable[i][0] > Maxx)
      Maxx =    FreqTable[i][0];
 int Maxxx = (int) (deltaX*Maxx + deltaX);
 //The maximum y-coordinate is found
 int Maxyy = 0;
 for ( int i = 0; i < n; i++ )
  if (FreqTable[i][1] > Maxyy)
       Maxyy = (int) FreqTable[i][1] + 2;
 //Starting clause
 //x- and y-units.
 //The needed space is defined
 System.out.println("\\begin{center}" +
     "\n\\psset{xunit=0.15,yunit=0.3}" +
 "\n\\begin{pspicture}(" + Minxxx +",0)(" +
 Maxxx+ "," + Maxyy+")");
 //A rectangle is drawn for each entry in the
 //frequency table

 int lim1,lim2;
 for ( int i = 0; i < n; i++ )
  {
   lim1 = (int) (deltaX*FreqTable[i][0] -deltaX/2);
   lim2 = (int) (lim1 + deltaX);
   System.out.print(
 "\n\\psline("+ lim1 + ",0)" +
     "("+lim1 +"," + FreqTable[i][1]+ ")" +
     "("+lim2 + ","+ FreqTable[i][1]+ ")" +
     "("+lim2 + ","+ "0)"
    );
  }

 //Horizontal axis
 System.out.println("\n\\psline(" +
      Minxxx +",0)("+ Maxxx + ",0)");

 //Class markers (first coordinates) are indicated
 for ( int i = 0; i < n; i++ )
   {
int xposition = (int) (deltaX* FreqTable[i][0]);
```

```
int yposition = - 1;
    System.out.print
 (
  "\n\\rput*("+ xposition + "," + yposition+ "){"+
   (int) (FreqTable[i][0])+ "}"
 );
   }
 System.out.println();

 //Labels for heights (second coordinates) are indicated
 for ( int i = 0; i < n; i++ )
  {
   int xposition = (int) (deltaX* FreqTable[i][0]);
   int yposition = (int) (1+ FreqTable[i][1]);
   System.out.print
(
  "\n\\rput*("+ xposition + "," + yposition+ "){" +
      (int) (FreqTable[i][1])+ "}"
 );
 }

 //End clause
 System.out.println( "\n\\end{pspicture}");
 System.out.println( "\\end{center}");
 }

public static void codePSTricks()
 {
  double  FreqT[][] =
        {{3, 3},
         {4, 3},
         {5, 2},
         {6, 2},
         {7, 1},
         {8, 1}};
 makeCode(FreqT);
 }
```

54. ***Exercise.*** *Add this method to your toolBox4 to create toolBox5. Play with the code.*

55. ***Challenge.*** *Imprint your personal seal over your toolBox by adding another function to analyze quantitative data, say, a method to sort data or to plot a steam and loaf diagram.*

2.4 Conclusion

There are too many styles for programming most possibly one for each developer. But given that designing software is very difficult, it is our duty to learn a style that possibly would alleviate that difficult task. Our election has been to single out a gossipless, transparent functional style of design. We propose that a style results from a strategy to solve problems so it is a direct result of having a rather limited mind. Therefore, we predict that evolution must have no style, i.e., so evolution must be dominated by disorder but locally it must follow a given style that never will get repeated anyhow. Moreover, we have illustrated one of our general claims: evolution of complex features must left a clear evolution towards perfection that must be apparent in the fossil record.

Chapter 3

The binomial distribution

`Coins and probabilities.`

Everyone has played with coins to decide who wins and who loses. So, we can use this background to ease the introduction of abstract concepts such as probability density functions.

56. ***Purpose.*** *To add to our toolbox some code that helps us to understand the binomial distribution. Since we recycle the material of the previous volume, we go rather rapidly.*

3.1 Patchwork style

Let us calculate the binomial distribution. We illustrate another style in which methods operate over reserved patches of memory like a farmer works and crops over the same area.

57. ***The code for binomial distribution. Style= farmer.***

```
//Program F57 binomial.
//This program outputs p(x)
//for given n and p,
//Style = farmer.

public class binomial
{
//Prints n data in vector
private static void printVector(double[] Vect, int n)
{
 //System.out.println( "Data are" );
 for ( int i = 0; i < n; i++ )
```

```
System.out.println( Vect[ i ]  );
}

//Sums data in vector with real numbers
private static double sumVector(double[] Vect, int n)
{
 double sum = 0;
 for ( int i = 0; i < n; i++ )
sum = sum + Vect[i];
 return sum;
}

//Returns numb!
private static long factorial (long numb)
{
 if (numb <=1 ) return 1;
 else
 return numb * factorial (numb -1);
}


//The  binomial distribution
//for  n repeats
//probability of head p,
//amplification factor F.
private static void  binomialDistribution(int n,
          double p, double F ,
          double[] CoefficientsBinomial,
          double[] pBinomial,
          double[] pBinomialF)
{
 long biCoeff = 0;
 for(int r = 0; r < n+1; r++)
  {
   biCoeff = factorial(n) /
       (factorial(r) * factorial(n-r));
   CoefficientsBinomial[r] = biCoeff;
  }
 System.out.println("\nCoefficients ");
 printVector(CoefficientsBinomial,n+1);
 System.out.println("Sum = " +
     sumVector(CoefficientsBinomial,n+1));

 for(int r = 0; r < n+1; r++)
  {
   pBinomial[r] = CoefficientsBinomial[r]  *
```

```
        Math.pow(p, r)* Math.pow(1-p, n-r);
  }
 System.out.println("\nProbabilities for 0,1,2...");
 System.out.println("Paste this vector to Excel" +
           " and make a graphic");
 printVector(pBinomial,n+1);
 sumVector(pBinomial,n+1);
 System.out.println("Sum = "
            +  sumVector(pBinomial,n+1));
 for(int k = 0; k < n+1; k++)
 pBinomialF[k] =  F* pBinomial[k];
 System.out.println("\nAbsolute frequencies for F = " + F);
 printVector(pBinomialF,n+1);
 System.out.println("Sum = "  +
           sumVector(pBinomialF,n+1));
}


//Detailed computation of the mean
private static void detailedMean(int n,
      double p,
      double[] pBinomial)
{
 System.out.println("\nDetailed mean: xp(x)");
 double  sum =0;
 double term;
 for(int r = 0; r < n+1; r++)
 {
  term =  r *  pBinomial[r];
  System.out.println( r + "    " + term) ;
  sum = sum +term;
 }
 System.out.println( "mean = sum xp(x) = " + sum) ;
 System.out.println( "Expected mean = np =  " + n*p) ;
}

//Detailed computation of the variance
private static void detailedVariance(int n,
      double p,
      double[] pBinomial)
{
 System.out.println("\nDetailed variance: " +
  "(x-mean)*(-mean)*p(x)");
 double  sum =0;
 double term;
 for(int r = 0; r < n+1; r++)
```

```
  {
   term =  Math.pow(r- n*p,2) *  pBinomial[r];
   System.out.println( r + "    " +  term) ;
   sum = sum +term;
  }
 System.out.println( "Variance = " +
"sum (x-mean)*(-mean)*p(x) = " + sum) ;
 System.out.println(
"Expected var = np(1-p) =  " + n*p*(1-p)) ;
 System.out.println( "Deviation = " + Math.sqrt(sum));
}

private static void binomialDist()
{
 System.out.println("BINOMIAL DISTRIBUTION ");
 double CoefficientsBinomial[] = new double[1000];
 double pBinomial[] = new double[1000];
 double pBinomialF[] = new double[1000];
 int n= 5;
 double p = 0.2;
 int F = 35;
 System.out.println("n= " + n);
 System.out.println("p = " + p);
 System.out.println("F = " + F);
 binomialDistribution(n,p, F ,CoefficientsBinomial,
                      pBinomial, pBinomialF);
 detailedMean(n,p,pBinomial);
 detailedVariance(n,p,pBinomial);
}

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

58. ***Exercise.*** *Run the program and play with the code.*

59. ***Exercise.*** *Pay attention to the method* `binomialDistribution(...)`*: it read inputs, n, p and F, then it makes some calculations whose output is located at pBinomial and other vectors but these already appear listed in the input. So, this style mixes input and output altogether. It resembles a farmer that seeds and crops in the same area. Rewrite the code following a gossipless, transparent, functional style.*

60. ***Exercise.*** *Compose the code to calculate $p(a \leq x \leq b)$ for a binomial distribution and add the code to the previous program.*

61. ***Challenge.*** *Compose the code to calculate for the binomial distribution the next quantities:*

a) $\Psi(a,b) = p(a < x < b)$*, probability of a closed interval;*

b) $\Psi[a,b] = p(a \leq x \leq b)$*, probability of an open interval;*

c) $\Psi[a,b) = p(a \leq x < b)$*, probability of a closed-open interval;*

d) $\Psi(a,b] = p(a < x \leq b)$*, probability of an open-closed interval;*

e) $\Psi[a] = p(x \leq a)$*, closed cumulative distribution function;*

f) $\Psi(a) = p(x < a)$*, open cumulative distribution function;;*

g) $\Psi[a >] = p(x \geq a)$*, closed survival function or p-value of a;*

h) $\Psi(a >) = p(x > a)$*, open survival function or p-value of a;*

62. ***Exercise.*** *For the binomial distribution, resolve the next inverse problem corresponding to finding the critical value associated to a given significance level* α*,* $0 \leq \alpha \leq 1$*: find the maximal* k *such that* $p(x \geq k) \geq \alpha$*. Formulate and solve the corresponding problem for the lower tail and next for two tails. Use the code to update binomial3 to binomial4.*

63. ***Exercise.*** *Use the code of binomial4 to update toolBox5 to version 6.*

3.2 Conclusion

We have worked probabilities, the cumulative distribution and critical values for the binomial distribution. We have shown how the farmer style functions and in this way we continue to support our claim that there are infinitely too many different styles of programming, possibly one for each developer.

Chapter 4

The normal and associated distributions

`Critical and p-values`

A normal distribution can be taken as an idealization of a distribution with central tendency that perfectly fits a binomial one. In statistical inference, one finds that the normal distribution is associated with other four, z, t, χ^2 and F, which conform the aim of this chapter.

64. ***Purpose.*** *Here we reuse the code of the previous volume to compose a program to calculate critical and p-values of the z, t, χ^2 and F distributions.*

4.1 The z distribution

A normal distribution with mean 0 and deviation 1 is the z**-distribution**. By a change of scale, on can use this distribution to study whatever normal one.

65. ***Definitions that make sense for continuous distributions of central tendency.***

1. *The* ***p-value*** *of an event e is $p(x \geq e)$, the probability of getting an event that is equal or greater than e. Events that are extreme, localized at the far reaches of the distribution, are interesting because their associated p-values are small, say less than 0.05.*

2. *The* ***significance level*** *α is a number $0 \leq \alpha \leq 1$ that represents the fraction of events that are considered as extreme and that deserve special studies to elucidate whether or not its existence is due to the same mechanism that generates those events at the center of the distribution. If the significance level is concentrated in one tail of the distribution, we say that we deal with* ***one tail*** *but if its divided in the two tails, we say that we deal with* ***two tails****.*

3. *The **upper critical value** associated to the upper tail and significance level α is an event x_+ such that its p-value is exactly equal to α. In that case we say that an **event is extreme** if it is equal or greater than x_+. The **lower critical value** associated to the lower tail and significance level α is an event x_- such that $p(x \leq x_-) = \alpha$. In that case we say that an event is extreme if it is equal or less than x_-. If we deal with two tails, one divides the significance in two parts and finds the corresponding critical values, one for the upper tail and the other for the lower one. Extremes events are defined accordingly.*

4. *The **null hypothesis** is that every event is generated by the same mechanism that gives rise to the distribution. The **alternate hypothesis** is that there are also other acting mechanisms that eventually might interfere.*

5. *A **test** decides which hypothesis is recommendable to take. The decision taken by default is the following: if an event is classified as normal, not extreme, we accept the null hypotheses else we reject it and we advise to consider another possibly mechanism to explain observed event apart from the one that generated the observed distribution.*

6. *The scientific method is error prone: an event that is extreme might have been produced by the very same mechanisms that explain the considered distribution: this causes an α-error. By the same token, an event can be classified as normal, not extreme, but it might have been produced by alien mechanisms not considered in the computation of the studied distribution: we have a β-error.*

7. *This protocol defines the **naked scientific method**. Every person likes to dress the naked scientific method in an appropriate for her or him philosophical or religious ideology.*

66. *Exercise.* *Show that the probability of committing an error is α-dependent and find the probability of committing an α error. Do the same for β-errors.*

4.2 p-values

The code that calculates the p-values for our distributions, one by one, follows:

67. *The next code calculates the p-value for the z-distribution:*

```
//Program F67 pValueZ
//Reports  the p- value associated to a given event
//for a standardized normal distribution.
//Combines Simpson's rule
//with a change of scale to
//swiftly calculate the integral
//under the bell in within 0 and   z for any z.
```

```
public class pValueZ
{
 private static final double  pi = 3.14159265358979323846;

//Returns the transformed density function of the
//standardized normal distribution.
private static double zFunction(double t)
 {
  double z = t/(1-t*t);
  double  f = 1 /Math.pow(2*pi, 0.5)*Math.exp(-z*z/2)*
             (1+ t*t)/( (1-t*t)*(1-t*t));
  return f;
 }

//To use this method for another distribution,
//instead of  zFunction(z), write and make a call for your
//new distribution.
private static double f(double xValue)
 {
  return  zFunction(xValue);
 }

//Returns the integral under the function f
//in within 0 and z. Precision=1/N.
//Simpson's rule adapted for reuse.
private static double HalfBodySimpson(double z, long N)
 {
  double h = z/(2*N);
  long m = N;
  double sum = f(0);
  for(int j= 1; j<=m; j++)
 sum = sum +   4* f((2*j-1) * h);
  for(int j= 1; j<m; j++)
 sum = sum +   2* f((2*j) * h);
  sum = sum + f((2*m) * h);
  sum = (h/3)*sum;
  return sum;
 }


//The area in within zero and z of chosen
//distribution is calculated
//for various degrees of precision.
private static  double area(double z)
```

```
 {
  z = (Math.sqrt(1+4*z*z)-1)/(2*z);
  int N;
  double  answer;
  int i = 3;
  N = (int) Math.pow(10, i);
  answer = HalfBodySimpson(z,N);
  return answer;
 }

//Returns Psi(e) = p( x < e)
//the cumulative distribution function of z
private static  double psi(double z)
{
 return 0.5 + area(z);
}

//Returns the p-value(e) = p( x > e)
//the p-value of  z
private static  double pValue(double z)
{
 return 1- psi(z);
}


public static void main(String[] args)
 {
  System.out.println("\nz distribution");
  double testValue = 1.23456789;
  System.out.println("\nTest value = " + testValue);
  System.out.print("Simpson's rule."+
  "\nArea = p( 0 <  x < " + testValue + ") = ");
  double area = area(testValue);
  System.out.println( area);
  double twoAreas = 2* area;
  System.out.println("Two Areas = " + twoAreas);
  //psi(e) = p( x < e)
  double psi = psi(testValue);
  System.out.println("psi(" + testValue + ") " +
      "= p( x < " + testValue + ") = " +  psi );
  double pValue = pValue(testValue);
  System.out.println("p-value(" +
      testValue + ") = " +
    "p( x > " + testValue + ") = " + pValue);
 }
}
```

68. *Exercise.* *Run the program and play with the code. Check the precision of the code with Gnumeric.*

69. *Exercise.* *Reuse the previous code to devise a software to calculate p-values for the t-distribution. Check the precision of the code with Gnumeric. The density function of the t- distribution is*

$f_t(x) = \frac{\Gamma[(\nu+1)/2]}{\Gamma(\nu/2)\sqrt{\nu\pi}}(1+\frac{x^2}{\nu})^{-\frac{\nu+1}{2}}$

where the Γ *function is defined by*

$$\Gamma(x/2) = \begin{cases} (x/2-1)(x/2-2)...3\times 2\times 1 \ if \ x \ is \ even. \\ (x/2-1)(x/2-2)...3/2\times 1/2\times\sqrt{\pi} \ if \ x \ is \ odd. \end{cases}$$

70. *Exercise.* *Reuse the previous code to devise a software to calculate p-values for the* χ^2*-distribution. Keep in mind that the density function of this distribution is zero below zero. Check the precision of the code with Gnumeric. The probability density function of the* χ^2*- distribution is*

$f_{\chi^2}(x) = \frac{1}{2^{\nu/2}\Gamma(\nu/2)}e^{-\frac{x}{2}}\,x^{\frac{\nu-2}{2}}$ *for* $x \geq 0$.

71. *Exercise.* *Reuse the previous code to devise a software to calculate p-values for the F-distribution. Keep in mind that the density function of this distribution is zero below zero. Check the precision of the code with Gnumeric. The probability density function is given by*

$f_F(x) = \frac{\Gamma(\frac{\nu_1+\nu_2}{2})}{\Gamma(\frac{\nu_1}{2})\Gamma(\frac{\nu_2}{2})}\nu_1^{\frac{\nu_1}{2}}\nu_2^{\frac{\nu_2}{2}}\ \frac{x^{\frac{\nu_1^{-2}}{2}}}{(\nu_2+\nu_1 x)^{\frac{\nu_1+\nu_2}{2}}}$ *for* $x \geq 0$.

where Γ *is the gamma function defined in the chi-square theorem, the degrees of freedom are* $\nu_1 = n_1 - 1$ *and* $\nu_2 = n_2 - 1$.

4.3 Merging pieces into a unity

We have various programs to calculate the p-values of diverse distributions. A next purpose is to add it to our toolbox. We can do that in various ways, the very first is just to add one class after another in tandem.

The problem with this option is that it is not elegant because all programs have a lot in common while the main difference lies in the methods that define the specific distribution and some other helping methods.

To produce a single class without redundancy one might include a lot of ifs that discriminate which distribution must be dealt with. Now, when we have various ifs, we have the option to use the Java switch structure as follows:

72. *The switch structure*

One uses the switch structure when one is presented with a multiple selection problem as follows:

```
// switch multiple selection structure
 public static void permutator(int k)
 {
       switch (k)
{
case 0: zero(); break;
case 1: one(); break;
case 2: two(); break;
case 3: three(); break;
case 4: four(); break;
case 5: five(); break;
case 6: six(); break;
case 7: seven() ; break;
case 8: eight(); break;
case 9: nine(); break;
}
 }
```

This structure is executed by reading the value of k and executing the corresponding case, if any, and the structure is abandoned in the next step. The program with the complete code follows:

```
//Program  F72 switchProg
//to learn the switch structure:
//it functions like a   multiple if-structure.
//No slack variables.
//How to   permute methods.
import java.util.Random;


public class switchProg
{
//Turn on of the random generator
static Random r  = new Random();

public static void zero()
{
 System.out.print(0);
}
```

```
public static void one()
{
 System.out.print(1);
}

public static void two()
{
 System.out.print(2);
}

public static void three()
{
 System.out.print(3);
}

public static void four()
{
 System.out.print(4);
}

public static void five()
{
 System.out.print(5);
}

public static void six()
{
 System.out.print(6);
}

public static void seven()
{
 System.out.print(7);
}

public static void eight()
{
 System.out.print(8);
}

public static void nine()
{
 System.out.print(9);
}
```

```
public static void other()
{
 System.out.print("Not implemented");
}

// switch multiple selection structure
 public static void permutator(int k)
 {
       switch (k)
{
case 0: zero(); break;
case 1: one(); break;
case 2: two(); break;
case 3: three(); break;
case 4: four(); break;
case 5: five(); break;
case 6: six(); break;
case 7: seven() ; break;
case 8: eight(); break;
case 9: nine(); break;
default: other(); break;
}
 }


public static   void main(String[] args)
    {
 System.out.println("My number");
 permutator(7);
 System.out.println("\n\nVoid number");
 permutator(14);
 System.out.println("\n\nDigits");
     for(int k=0; k<=9; k++) permutator(k);
     System.out.println("\n\nDigits in backward order");
     for(int k=0; k<=9; k++) permutator(9-k);
     System.out.println("\n\nRandom digits");
     for(int k=0; k<=9; k++) permutator(r.nextInt(10));
    }
}
```

73. ***Exercise.*** *Run the program and play with the code.*

74. ***Challenge.*** *Use many ifs else a switch structure to try to merge all programs that calculate p-values into a single simplified program that executes all the tasks of all programs. If you succeed, crown yourself with gold: that is more difficult than getting to the moon.*

75. ***The OOP strategy***

We have another option to merge our various programs that calculate p-values for different distributions. It is a powerful generalization of the next program:

76. ***The code for a very simple program.***

```
//Program F76 simpleCode
public class simpleCode
{
 public static void main(String[] args)
  {
int r;
r=5;
System.out.println(" r  = " + r);
r=57;
System.out.println(" r  = " + r);
}
}
```

We can interpret this code as a solution to the problem of having Java to write diverse constants using the same printing instruction. Now, if Java can do this with constants, we shall ask: Is Java able to do the same but not with constants but with methods and eventually with complete classes?

The answer is affirmative: the designers of Java implemented this possibility in two forms: interfaces and abstract classes. We will see here the second implementation.

77. ***Abstract classes***

Please, forget about this terrible name and fix your attention of the problem to solve: we must simplify redundant code that execute the same task but over different objects. The task is to integrate a function. The method that executes the integration is common for all functions but the method that defines the function varies over a given set of allowed functions.

The Java solution is as follows:

1. Take a program that executes the task over a given object. Detect what is the common part to all programs and what is the varying part.
2. Common methods remain in the class but varying methods are just listed but not implemented.
3. Varying methods are implemented in special classes.
4. There must be a manager class, usually the main class, that directs all the work.

Let us execute this program over a toy model.

78. ***The code for a case of integration.***

Our task is to produce a code to integrate in within zero and 1 specific members of two types of functions. The first is the family of straight lines $y = a + bx$, and the second is $y = ax^b$. Both families depend on two parameters, a and b. The idea of integration is of course the very same for both families but we must adapt the general procedure to the specific circumstances. This problem is very easy since we know how to integrate those functions analytically:

$\int_0^1 (a + bx)dx = (ax + bx^2/2)|_0^1 = a + b/2.$
$\int_0^1 (ax^b)dx = (ax^{b+1}/(b + 1))|^1{}_0 = a/(b + 1).$

Let us put this formula into a code which must not be redundant:

```
//Program F78 AbstractIntegration
//A program  that integrates functions
//is encoded.
//Interval of integration from 0 to 1.
//Two functions are allowed:
//a line and a monomial.


public class AbstractIntegration
{

//Outer class is instantiated
static  AbstractIntegration a =new AbstractIntegration();

//Inner top class that defines commonalities to all tasks
abstract class Function
{
double a;
double b;

//How a Function is initialized
Function(double c1, double c2)
{
a = c1;
b = c2;
}

// Integration depends on the type of function
//and is not implemented, it is just listed.
abstract double integral();
```

```
}//end of class Function


//Inner class that defines a particular implementation
//of the abstract class Function
class Line extends Function
 {
  //How a line is initialized
  //Coefficients of the line
  // a +bx
  Line(double a, double b)
   {
    //The same as the in the top class
    super(a, b);
   }


  //The integral for a line is specified
  double integral() {
  System.out.print("integral of " + a + " + " +
           b + "x in within 0 and 1  = ");
  return  a + b/2;
  }
 }//end of class Line

class Monomial extends Function
 {
  //How a monomial is initialized
  //Coefficient and power of the monomial
  //axˆb
  Monomial(double a, double b)
   {
//The same as  in the top class
    super(a, b);
   }

  //The integral for a monomial is specified
  double integral()
  {
   System.out.print("integral of " + a + "xˆ" +
            b +" in within 0 and 1  = ");
   return a/(b+1);
  }
}//end of class monomial.
```

```
//General manager:
//the very same instruction processes different functions
//that can be chosen at will.
public static void main(String args[])
{
 Line l = a.new Line(9, 5);//9 + 5x
 Monomial m = a.new Monomial(10, 8); //10xˆ8
 Function f; //Abstract instantiation of Function into f
 f = l; //instantiation of f
 System.out.println(  f.integral());
 f = m; //instantiation of f
 System.out.println( f.integral());
}
}
```

79. ***Exercise.*** *Run the program and play with the code.*

The difference of this problem and the one we have with various distributions is just that we have no analytical formulas for their integration and must proceed numerically, a hard work made in the previous volume. So, let us proceed step by step. The fist one is just to recast the code that calculate p-values for the z-distribution in the new language of abstract classes.

80. ***The code that uses abstract classes to calculate p-values of the z-distribution follows:***

```
//Program F80 pValues1
//Reports   p- value
//for the z-distribution.
//Combines Simpson's rule
//with a change of scale.
//Code is recast in the language of abstract classes.
//This is evolution to the full:
//one must get powerful but with a complexity that
//is perfect  to resolve specific problems.


public class pValues1
{

 //Outer class is instantiated
 static pValues1 a = new pValues1();

 //This is constant pi
 private static final double  pi = 3.14159265358979323846;
```

```
//======Inner top class with commonalities=========
//The methods that eventually can be useful
//to other distributions come here such as they are.
//The methods that are privative of the z distribution
//are defined here as abstract
//and implemented in a specific classes below.
abstract class pValueAbstract
{

 //The function that defines z is replaced
 //by an abstract type. It is not implemented,
 //so we can implement other functions too and
 //process them with this very same machinery.
 abstract double Function(double t);


//To use this method for another distribution,
//instead of  Function(z), write and make a call for your
//new distribution.
double f(double xValue)
 {
  return  Function(xValue);
 }

//Returns the integral under the function f
//in within 0 and z. Precision=1/N.
//Simpson's rule adapted for reuse.
double HalfBodySimpson(double z, long N)
 {
  double h = z/(2*N);
  long m = N;
  double sum = f(0);
  for(int j= 1; j<=m; j++)
 sum = sum +   4* f((2*j-1) * h);
  for(int j= 1; j<m; j++)
 sum = sum +   2* f((2*j) * h);
  sum = sum + f((2*m) * h);
  sum = (h/3)*sum;
  return sum;
 }


//The area in within zero and z of chosen
//distribution is calculated
//for various degrees of precision.
```

```
double area(double z)
 {
  z = (Math.sqrt(1+4*z*z)-1)/(2*z);
  int N;
  double  answer;
  int i = 3;
  N = (int) Math.pow(10, i);
  answer = HalfBodySimpson(z,N);
  return answer;
 }

//Returns Psi(e) = p( x < e)
//the cumulative distribution function of e.
//This is abstract because it depends
//on the distribution, it must be
//implemented in the corresponding classes.
abstract double psi(double e);


//Returns the p-value(e) = p( x > e)
//Local implementation of the p-value of  e
double pValue(double e)
{
 return 1- psi(e);
}


//Local manager
void executor(double testValue)
{
 System.out.println("\nTest value = " + testValue);
 System.out.print("Simpson's rule."+
  "\nArea = p( 0 <  x < " + testValue + ") = ");
 double area =  area(testValue);
 System.out.println( area);
 //psi(e) = p( x < e)
 double psi = psi(testValue);
 System.out.println("psi(" + testValue + ") " +
    "= p( x < " + testValue + ") = " +  psi );
 double pValue = pValue(testValue);
 System.out.println("p-value(" +
      testValue + ") = " +
    "p( x > " + testValue + ") = " + pValue);
}

}//end of class pValueAbstract
```

```
//======Particular class========

//Definition of the z density function
//and exclusive to it methods.
//It is an specification of the pValueAbstract class.
class pVzClass extends pValueAbstract
{

 //Constructor: this is the first thing to do
 pVzClass()
 {
  System.out.println("\nz distribution");
 }

//Returns Psi(e) = p( x < e)
//the cumulative distribution function of e
double psi(double e)
{
 return 0.5 + area(e);
}

//Function is implemented here.
//Returns the transformed density function of the
//standardized normal distribution.
double Function(double t)
{
 double z = t/(1-t*t);
 double  f = 1 /Math.pow(2*pi, 0.5)*Math.exp(-z*z/2)*
             (1+ t*t)/( (1-t*t)*(1-t*t));
 return f;
}

}//end of class zClass



//General manager:
//We get prepared to use
//the very same instruction
//to process different distributions
//that can be chosen at will.
public static void main(String args[])
{
  double testValue = 1.23456789;
```

```
  //Particular class is instantiated
  pVzClass z = a.new pVzClass();
  //Top class is declared
  pValueAbstract p;
  //Top class is instructed to take
  //the form of the particular class
  p = z;
  //Asked task is made
  p.executor(testValue);
 }

}//End of main class
```

81. ***Exercise.*** *Run the program and play with the code.*

Let us see how can we add the capability to process a new distribution.

82. ***The code augmented to process the t-distribution follows:***

```
//Program F82 pValues2
//Reports   p- value
//for the z and t distributions.
//Combines Simpson's rule
//with a change of scale.
//Code is recast in the language of abstract classes.
//This is evolution to the full:
//one must get powerful but with a complexity that
//is perfect  to resolve specific problems.


public class pValues2
{

 //Outer class is instantiated
 static pValues2 a = new pValues2();

 //This is constant pi
 private static final double  pi = 3.14159265358979323846;


//====== Methods of general use=========

//Returns numb!
private static long factorial (long numb)
 {
  if (numb <=1 ) return 1;
else
```

```
  return numb * factorial (numb -1);
 }


//Returns the double factorial
private static double doubleFact(double x)
{
if (x <=0 ) return 1;
else
 return x * doubleFact(x -2);
}


//Returns the Gamma function
private static double Gamma(double x)
{
 int xRounded = (int) x;
 //System.out.println( x + "   " + xRounded);
 if (xRounded == x)
return factorial (xRounded-1);
 else
  {
 return Math.sqrt(pi)* doubleFact(2*xRounded -1) /
          Math.pow(2,xRounded);
  }
}


//======Inner top class with commonalities=========
//The methods that eventually can be useful
//to other distributions come here such as they are.
//The methods that are privative of each distribution
//are defined here as abstract
//and implemented in  specific classes below.
abstract class pValueAbstract
{
 //Degrees of freedom of the t distribution
 double dfT;

 //Acquisition of data (abstract declaration)
 pValueAbstract(double a)
 {
  dfT = a;
 }

 //The function that defines z is replaced
```

```
 //by an abstract type. It is not implemented,
 //so we can implement other functions too and
 //process them with this very same machinery.
 abstract double Function(double t);


//To use this method for another distribution,
//instead of  Function(z), write and make a call for your
//new distribution.
double f(double xValue)
 {
  return  Function(xValue);
 }

//Returns the integral under the function f
//in within 0 and z. Precision=1/N.
//Simpson's rule adapted for reuse.
double HalfBodySimpson(double z, long N)
 {
  double h = z/(2*N);
  long m = N;
  double sum = f(0);
  for(int j= 1; j<=m; j++)
 sum = sum +   4* f((2*j-1) * h);
  for(int j= 1; j<m; j++)
 sum = sum +   2* f((2*j) * h);
  sum = sum + f((2*m) * h);
  sum = (h/3)*sum;
  return sum;
 }


//The area in within zero and z of chosen
//distribution is calculated
//for various degrees of precision.
double area(double z)
 {
  z = (Math.sqrt(1+4*z*z)-1)/(2*z);
  int N;
  double  answer;
  int i = 3;
  N = (int) Math.pow(10, i);
  answer = HalfBodySimpson(z,N);
  return answer;
 }
```

```
//Returns Psi(e) = p( x < e)
//the cumulative distribution function of e.
//This is abstract because it depends
//on the distribution, it must be
//implemented in the corresponding classes.
abstract double psi(double e);


//Returns the p-value(e) = p( x > e)
//Local implementation of the p-value of  e
double pValue(double e)
{
 return 1- psi(e);
}


//Local manager
void executor(double testValue)
{
 System.out.println("\nTest value = " + testValue);
 System.out.print("Simpson's rule."+
  "\nArea = p( 0 <  x < " + testValue + ") = ");
 double area =  area(testValue);
 System.out.println( area);
 //psi(e) = p( x < e)
 double psi = psi(testValue);
 System.out.println("psi(" + testValue + ") " +
    "= p( x < " + testValue + ") = " +  psi );
 double pValue = pValue(testValue);
 System.out.println("p-value(" +
      testValue + ") = " +
    "p( x > " + testValue + ") = " + pValue);
}

}//end of class pValueAbstract


//======Particular classes========


//The z class.
//Definition of the z density function
//and exclusive to it methods.
//It is an specification of the pValueAbstract class.
class pVzClass extends pValueAbstract
{
```

```
 //Constructor: this is the first thing to do
 pVzClass(double dfT)
 {
  //Coordinate values with the top class
  super(dfT);
  System.out.println("\nz distribution");
 }

//Returns Psi(e) = p( x < e)
//the cumulative distribution function of e
double psi(double e)
{
 return 0.5 + area(e);
}

//Function is implemented here.
//Returns the transformed density function of the
//standardized normal distribution.
double Function(double t)
{
 double z = t/(1-t*t);
 double  f = 1 /Math.pow(2*pi, 0.5)*Math.exp(-z*z/2)*
             (1+ t*t)/( (1-t*t)*(1-t*t));
 return f;
}

}//end of class pVzClass


//The t class
//Definition of the t density function
//and exclusive to it methods.
//It is an specification of the pValueAbstract class.
class pVtClass extends pValueAbstract
{

//Constructor: this is the first thing to do
pVtClass(double dfT)
{
 //Coordinate values with the top class
 super(dfT);
 System.out.println("\nt distribution");
 System.out.println("t degrees of freedom = " + dfT);
}
```

```
//Returns Psi(e) = p( x < e)
//the cumulative distribution function of z
double psi(double z)
{
 return 0.5 + area(z);
}


//Returns the transformed density function of the
//t distribution with dfT  d.f.
double Function(double t)
{
 double nu = dfT;
 double z = t/(1-t*t);
 double  f = ( Gamma((nu+1)/2) /
 ( Math.sqrt(pi*nu)*Gamma(nu/2)) )
             * Math.pow(1+z*z/nu, -(nu+1)/2 )
             * (1+ t*t)/( (1-t*t)*(1-t*t));
 return f;
}

}//end of class pVtClass



//General manager:
//We  use
//the very same instruction
//to process different distributions
//that can be chosen at will.
public static void main(String args[])
{
 double testValue = 1.23456789;
 //Degrees of freedom of the t distribution
 double dfT = 9;
 //Top class is declared
 pValueAbstract p;

 //Particular classes  are instantiated
 pVzClass z = a.new pVzClass(dfT);
 //Top class is instructed to take
 //the form of the particular class
 p = z;
 //Asked task is made
 p.executor(testValue);
```

```
 pVtClass t = a.new pVtClass(dfT);
 p = t;
 //Asked task is made
 p.executor(testValue);
}

}//End of main class
```

83. ***Exercise.*** *Run the program and play with the code.*

84. ***Exercise.*** *Add to the previous code the possibility to process the χ^2 distribution. Test your program with Gnumeric.*

85. ***Exercise.*** *Add to the previous code the possibility to process the F distribution. Test your program with Gnumeric.*

86. ***Overriding.*** *Our purpose in this section was to eliminate repetitions when various programs that execute the same task but over different objects must be merged into a single unity. To that aim, we used abstract classes. Anyway, there are many forms as this can be done. In particular, the style developed by the Author in given examples and answers is as yet redundant: the psi(..) method has 4 versions, one in each particular class, but they are repetitions of only two items. In the general case, the cheapest solution to this problem is to implement in the abstract class the most frequent version and to redefine locally the correct version when this contradicts that of the abstract class: redefinitions in particular classes override those in the top class if these are not static. The corresponding implementation follows:*

```
//Program F86 pValues5
//Reports   p- value
//for the z, t, chi2 and F distributions.
//Combines Simpson's rule
//with a change of scale.
//Code is recast in the language of abstract classes.
//This is evolution to the full:
//one must get powerful but with a complexity that
//is perfect  to resolve specific problems.


public class pValues5
{

 //Outer class is instantiated
 static pValues5 a = new pValues5();

 //This is constant pi
```

```
 private static final double  pi = 3.14159265358979323846;


//====== Methods of general use=========

//Returns numb!
private static long factorial (long numb)
 {
  if (numb <=1 ) return 1;
else
  return numb * factorial (numb -1);
 }


//Returns the double factorial
private static double doubleFact(double x)
{
if (x <=0 ) return 1;
else
 return x * doubleFact(x -2);
}


//Returns the Gamma function
private static double Gamma(double x)
{
 int xRounded = (int) x;
 //System.out.println( x + "   " + xRounded);
 if (xRounded == x)
return factorial (xRounded-1);
 else
  {
 return Math.sqrt(pi)* doubleFact(2*xRounded -1) /
          Math.pow(2,xRounded);
  }
}



//======Inner top class with commonalities=========
//The methods that eventually can be useful
//to other distributions come here such as they are.
//The methods that are privative of each distribution
//are defined here as abstract
//and implemented in  specific classes below.
abstract class pValueAbstract
```

```
{
 //Degrees of freedom of various distributions
 double dfT;
 double dfChi2;
 double dfFNum;
 double dfFDen;

//Acquisition of data (abstract declaration)
 pValueAbstract(double a, double b, double c, double d)
 {
  dfT = a;
  dfChi2 = b;
  dfFNum = c;
  dfFDen = d;
 }

 //The function that defines z is replaced
 //by an abstract type. It is not implemented,
 //so we can implement other functions too and
 //process them with this very same machinery.
 abstract double Function(double t);



//Function might be whatever
double f(double xValue)
 {
  return  Function(xValue);
 }

//Returns the integral under the function f
//in within 0 and z. Precision=1/N.
//Simpson's rule adapted for reuse.
double HalfBodySimpson(double z, long N)
 {
  double h = z/(2*N);
  long m = N;
  double sum = f(0);
  for(int j= 1; j<=m; j++)
 sum = sum +   4* f((2*j-1) * h);
  for(int j= 1; j<m; j++)
 sum = sum +   2* f((2*j) * h);
  sum = sum + f((2*m) * h);
  sum = (h/3)*sum;
  return sum;
 }
```

```
//The area in within zero and z of chosen
//distribution is calculated
//for various degrees of precision.
double area(double z)
 {
  z = (Math.sqrt(1+4*z*z)-1)/(2*z);
  int N;
  double  answer;
  int i = 3;
  N = (int) Math.pow(10, i);
  answer = HalfBodySimpson(z,N);
  return answer;
 }

//Returns Psi(e) = p( x < e)
//the cumulative distribution function of e.
//This is abstract because it depends
//on the distribution, it must be
//implemented in the corresponding classes.

//Returns Psi(e) = p( x < e)
//the cumulative distribution function of e
double psi(double e)
{
return 0.5 + area(e);
}

//Returns the p-value(e) = p( x > e)
//Local implementation of the p-value of  e
double pValue(double e)
{
 return 1- psi(e);
}


//Local manager
void executor(double testValue)
{
 System.out.println("\nTest value = " + testValue);
 System.out.print("Simpson's rule."+
  "\nArea = p( 0 <  x < " + testValue + ") = ");
 double area =  area(testValue);
 System.out.println( area);
 //psi(e) = p( x < e)
```

```
 double psi = psi(testValue);
 System.out.println("psi(" + testValue + ") " +
    "= p( x < " + testValue + ") = " +  psi );
 double pValue = pValue(testValue);
 System.out.println("p-value(" +
      testValue + ") = " +
    "p( x > " + testValue + ") = " + pValue);
}

}//end of class pValueAbstract


//======Particular classes========


//The z class.
//Definition of the z density function
//and exclusive to it methods.
//It is an specification of the pValueAbstract class.
class pVzClass extends pValueAbstract
{

 //Constructor: this is the first thing to do
 pVzClass(double dfT, double dfChi2,
  double dfFNum, double dfFDen)
 {
  //Coordination with the top class
  super(dfT, dfChi2, dfFNum,  dfFDen);
  System.out.println("\nz distribution");
 }


//Function is implemented here.
//Returns the transformed density function of the
//standardized normal distribution.
double Function(double t)
{
 double z = t/(1-t*t);
 double  f = 1 /Math.pow(2*pi, 0.5)*Math.exp(-z*z/2)*
             (1+ t*t)/( (1-t*t)*(1-t*t));
 return f;
}

}//end of class pVzClass
```

```
//The t class
//Definition of the t density function
//and exclusive to it methods.
//It is an specification of the pValueAbstract class.
class pVtClass extends pValueAbstract
{

//Constructor: this is the first thing to do
pVtClass(double dfT,double dfChi2,
 double dfFNum, double dfFDen)
{
 //Coordination with the top class
 super(dfT,dfChi2,  dfFNum,  dfFDen);
 System.out.println("\nt distribution");
 System.out.println("t degrees of freedom = " + dfT);
}



//Function is implemented here.
//Returns the transformed density function of the
//t distribution with dfT d.f.
double Function(double t)
{
 double nu = dfT;
 double z = t/(1-t*t);
 double  f = ( Gamma((nu+1)/2) /
       ( Math.sqrt(pi*nu)*Gamma(nu/2)) ) *
 Math.pow(1+z*z/nu, -(nu+1)/2 )  * (1+ t*t)/( (1-t*t)*(1-t*t));
 return f;
}

}//end of class pVtClass



//The chi2 class
//Definition of the Chi2 density function
//and exclusive to it methods.
//It is an specification of the pValueAbstract class.
class pVChi2Class extends pValueAbstract
{

//Constructor: this is the first thing to do
pVChi2Class(double dfT,double dfChi2,
    double dfFNum, double dfFDen)
```

```
{
 //Coordination with the top class
 super(dfT,dfChi2, dfFNum,  dfFDen);
 System.out.println("\nChi2 distribution");
 System.out.println("Chi2 degrees of freedom = " + dfChi2);
}

//Returns Psi(e) = p( x < e)
//the cumulative distribution function of e
double psi(double e)
{
 return area(e);
}

//Function is implemented here.
//Returns the transformed density function of the
//chi2 distribution with nu d.f.
double Function(double t)
{
 double nu = dfChi2;
 double z = t/(1-t*t);
 double  f = ( 1 /
   ( Math.pow(2, nu/2)*Gamma(nu/2)) )
   * Math.exp(-z/2)* Math.pow(z, (nu-2)/2)
   * (1+ t*t)/( (1-t*t)*(1-t*t));
 return f;
}


}//end of class pVChi2Class



//The F class
//Definition of the Chi2 density function
//and exclusive to it methods.
//It is an specification of the pValueAbstract class.
class pVFClass extends pValueAbstract
{

//Constructor: this is the first thing to do
pVFClass(double dfT,double dfChi2,
    double dfFNum, double dfFDen)
{
//Coordination with the top class
super(dfT,dfChi2, dfFNum,  dfFDen);
```

```
System.out.println("\nF distribution");
System.out.println("F degrees of freedom: "
           + "Num = " + dfFNum
           + ", Den = " + dfFDen);
}

//Returns Psi(e) = p( x < e)
//the cumulative distribution function of e
double psi(double e)
{
return area(e);
}

//Function is implemented here.
//Returns the transformed density function of the
//F distribution with dfFNum and dfFDen d.f.
double Function(double t)
{
double nu1 = dfFNum;
double nu2 = dfFDen;
double z = t/(1-t*t);
double  f = ( Gamma((nu1+nu2)/2) /
       ( Gamma(nu1/2) *Gamma(nu2/2)  ) ) *
 Math.pow(nu1, nu1/2) * Math.pow(nu2, nu2/2) *
 Math.pow(z,   nu1/2-1)      /
 Math.pow( nu2+nu1*z, (nu1+nu2)/2)
 *(1+ t*t)/( (1-t*t)*(1-t*t));
return f;
}


}//end of class pVFClass


//General manager:
//We  use
//the very same instruction
//to process different distributions
//that can be chosen at will.
public static void main(String args[])
{
 double testValue = 1.23456789;
 //Degrees of freedom of the t distribution
 double dfT = 9;
 //Degrees of freedom of the chi2 distribution
 double dfChi2 = 17;
```

```
 //Degrees of freedom of the F distribution, numerator
 double dfFNum = 4;
 //Degrees of freedom of the F distribution, denominator
 double dfFDen = 32;
 //Top class is declared
 pValueAbstract p;

 //Particular class z is instantiated
 pVzClass z = a.new pVzClass(dfT, dfChi2, dfFNum,  dfFDen);
 //Top class is instructed to take
 //the form of the particular class
 p = z;
 //Asked task is made
 p.executor(testValue);
 //Particular class t is instantiated
 pVtClass t = a.new pVtClass(dfT, dfChi2, dfFNum,  dfFDen);
 p = t;
 p.executor(testValue);
 //Particular class chi2 is instantiated
 pVChi2Class chi2 =
       a.new pVChi2Class(dfT, dfChi2, dfFNum,  dfFDen);
 p = chi2;
 p.executor(testValue);
 //Particular class F is instantiated
 pVFClass f =
     a.new pVFClass(dfT, dfChi2, dfFNum,  dfFDen);
 p = f;
 p.executor(testValue);

 //Test:
 //p-values of 1.23456789 by Gnumeric:
 //z: 0.10849568277116
 //t: 0.12412629591734
 //chi2: 0.99999991994117
 //F: 0.31591556865765
}

}//End of main class
```

87. ***Exercise.*** *Run the program and play with the code.*

4.4 Critical values

Our next task is to produce the code to calculate critical values for a given significance level for the various distributions.

88. ***The code that outputs critical values for the z-distribution follows.***

```
//Program F88 critZ
//Reports   critical values
//for a standardized normal distribution
//and a given significance level.
//Combines Simpson's rule
//with a change of scale.
//An initial approximation is improved
//by analytical hunting.
//Program is encapsulated in an inner class.
//Style = Entangled OOP.
public class critZ
{

//Outer class is instantiated
static critZ a = new critZ();


//===========Inner class ==========

//Reports   critical values
//for the z-distribution
class critZInner
{
 double  pi = 3.14159265358979323846;
 double Error;
 double jCritic;


 //==========Style: reuse===========

 //Returns the transformed density function of
 //the standardized normal distribution.
double zFunction(double t)
 {
  double z = t/(1-t*t);
  double  f = 1 /Math.pow(2*pi, 0.5)*Math.exp(-z*z/2)*
             (1+ t*t)/( (1-t*t)*(1-t*t));
  return f;
 }

 //To use this method for another distribution,
 //instead of  zFunction(z), write and make a call for your
 //new distribution.
double f(double x)
```

```
 {
  double  f = zFunction(x);
  return f;
 }

//Reports the transformed upper u-critical for
//the given distribution.
//Significance level = alpha.
//Precision=1/N.
//Simpson's rule adapted for reuse.
//The error of approximation is also calculated.
double upperU(double h, double target)
 {

  double r = h/6;
  double sum = 0.5;
  jCritic = 0;
  for(int j= 0; sum < target; j++)
   {
 sum = sum +
     r*(f( j*h) + 4* f((j+0.5) * h) +  f((j+1) * h));
 jCritic = j+1;
   }
  jCritic = jCritic-1;
  double greatSum = sum -
          r*(f( jCritic*h) + 4* f((jCritic+0.5) * h)
         +  f((jCritic+1) * h));
  Error = target-greatSum;
  double uAprox = (jCritic) *h;
  return uAprox;
 }

//Returns the transformed value z = u/(1-u*u)
double z(double u)
 {
  double z = u/(1-u*u);
  return z;
 }


//The wide of subintervals is defined
double h()
{
  double u = 1;
  int i=3;
  long N = (int) Math.pow(10, i);
```

```
  double h = u/N;
  return h;
}

 //Reports the upper z-critical for
 //Significance level = alpha.
double vCrit(double target)
 {
  double h = h();
  double uCritic =  upperU(h, target);
  double zCritic = z(uCritic);
  return zCritic;
 }


 //======Analytical hunting========
 //Precision is augmented.

 //Returns the density function of the Z distribution
double ff(double z)
 {
  double  f =  Math.exp(-z*z/2)/Math.pow(2*pi, 0.5);
  return f;
 }


// Returns F(a,b) = h(f(a)  + 4 * f((a+b)/2)  + f(b))-Error;
double F(double a, double b)
 {
  double h = (b-a)/6;
  double  Fb = h* (  ff(a) + 4 * ff((a+b)/2)
                   + ff(b)  )-Error;
  return Fb;
 }


//Returns the  ratio  F(zCritic,gi)/ ff(gi)
double ratio(double zCritic, double gi)
 {
  double  ratio =  F(zCritic,gi)/ ff(gi) ;
  return ratio;
 }

//Critical value for the upper tail.
//A professional  approximation is found by
//analytical hunting
```

```
double hunting(double zCritic )
 {
  double h = h();
  double ur =  (jCritic+1)*h;
  //Right border of the critical interval
  double rBorder = z(ur);
  double gSubi = rBorder;
  double gSubiPlus= 0;
  int iterations = 0;
  int maxIterations = 30;
  boolean more = true;
  while (more)
   {
gSubiPlus = gSubi -  ratio(zCritic, gSubi);
iterations = iterations +1;
if (iterations >= maxIterations)  more = false;
if (gSubiPlus == gSubi) more = false;
gSubi = gSubiPlus;
   }
  return gSubiPlus;
 }

//Critical value for significance level alpha, one tail
double  critical(  double alpha)
{
 double target = 1-alpha;
 double zCritic = vCrit(target);
 zCritic = hunting(zCritic);
 return zCritic;
}

//Local manager
void managerZ(double alpha)
  {
System.out.println("\nZ distribution");
//alpha = 0.05;
System.out.println("Critical values for one tail "
    + "\nof the standardized normal "
+ "distribution.  "
+ "\nAlpha = Significance level = "
+ alpha + ".");

System.out.println("zCritic  = +/-"
+  critical(alpha) );
System.out.println("\nCritical values for two tails "
    + "\nof the standardized normal "
```

```
+ "distribution."
+ "\nAlpha = Significance level = "
+ alpha + ".");
double alphaHalf = alpha/2;
System.out.println("zCritic  = +/-"
            + critical(alphaHalf) );
  }
}

public static void main(String[] args)
{
 //Inner class is instantiated
 critZInner zc = a.new critZInner();
 //Significance level
 double alpha = 0.05;
 //critical z is required
 zc.managerZ(alpha);
}

//z = -1.64485362695147

}
```

89. ***Exercise.*** *Run the program and play with the code. Test its precision with Gnumeric.*

90. ***Style = entangled OOP.***

There are infinitely many styles of software development. The **entangled OOP style** encapsulates objects in, say, inner classes but admits global variables at their interior. One motivation for this can be grasped if we give a look at the next piece of code:

```
for(int j= 0; sum < target; j++)
 {
  sum = sum +
        r*(f( j*h) + 4* f((j+0.5) * h) +  f((j+1) * h));
  jCritic = j+1;
 }
```

This code calculates two values, `sum` and `jCritic`. If we want to output both variables and remain faithful to the functional paradigm, we must devise another class, an object, with these two variables. But, this may look as too a delicate option for a very simple problem. So, we can opt for tolerating those variables as global ones in within the inner class. This is a perfectly valid option for OOP.

Another reason for tolerating global variables is that we deal with functions that depend on parameters. The natural simplicity of proposed tasks lost its

transparency if we overburden the text with their explicit printing. So, it looks wise to work with them as global variables to conserve simplicity.

91. ***Exercise.*** *Compose a program that outputs critical values for the t-distribution. Test precision with Gnumeric.*

92. ***Exercise.*** *Compose a program that outputs critical values for the Chi2-distribution. Test precision with Gnumeric.*

93. ***Exercise.*** *Compose a program that outputs critical values for the F-distribution. Test precision with Gnumeric.*

94. ***Exercise.*** *Merge all previous four programs that calculates critical values into a single unity.*

95. ***Redundancy in the genome.***

We have tried to simplify redundant code but not to complete exhaustion. Why? The reason is that code is in general highly informative and when it is deprived of its redundancy, the density of information grows and so it becomes still more informative. This means that previous information is useless to predict what will come next. Therefore, the code becomes more random: high levels of information and randomness are equivalent. This is a mystery that might suggest that even the most complex information might appear by mere gambling.

Now, oversimplified code is just like hard rock because it is very difficult to restructure into another form. This means that oversimplified code is very difficult to correct in the case that it contains bugs. That is why it is not recommendable to oversimplify code unless thoroughly testing has been carried on.

Now, we can ask: is the genome simplified, or oversimplified?

Most probably not as the C-paradox compels us to think. The **C-paradox** is the observed fact that the size of the genome does not correlate with complexity. For instance, the size of the genome of half of gymnosperms species is larger than those of all mammalian species (Gregory, 2005). In general, the DNA content is usually constant in any one species while it can vary in different species by over a million-fold (Cavalier-Smith, 1985).

There are various reasons why the C-paradox is important. The most interesting for us is that evolution is expected to award simplification because of economic reasons. But there are are other forces trying to enlarge the genome, say, spontaneous duplications. Some aspects of the intricacies of the ensuing interaction can be read after Vol 1 of this series with some modifications:

We see that one can extend a satisfiable proposition in infinitely many varied forms and lengths if one adds sufficiently many new variables. This is a strong result that is not obvious: one could think that it is highly improbably to introduce new variables, new genes and functions, without disturbing the actual functions that enable life. Nevertheless, that can be done and can be done with

absolute certainty. In fact, recombination is a natural and permanent operation in nature, and it can lead to gene duplication. Thus, we think that expansive forces are always at work upon the genome, trying to enlarge it.

So, evolution would be similar to a female cat, whose neck has been adorned with a *pink strip*, and, since the strip is not annoying the cat in any sense, the cat goes around without taking care of the strip. The genome is predicted in this way to be overloaded with any sort of nascent genes, whose function is void or unknown because the cell lacks the substrate for they to work upon.

In ancient times, the prediction of an overloaded genome would be considered as verified with a simple and direct proof: in eukaryotes there is a code that is not transcribed, which is highly repetitive and that can be 50000 times as much as that which is transcribed into RNA. But today, it is anymore secure that the genome contains junk code: a given pseudo-gene (a modified copy of a functional gene that does not serve as template for producing a protein) was in ancient times considered junk DNA, but it is now thought to be important part of the gene regulative machinery (Lee, 2003). Additionally, according to Arneodo et al., (1996) that 'junk' DNA has a clear function: the folding of eukaryotic DNA into the cell nucleus involves a hierarchy of structures on different length scales that is determined by self-correlation in DNA, mainly of the so called junk type.

Thus, junk DNA is not a proof that the genome is redundant. At the other hand, we do know that a simplifying force is under operation, as a study of parasites shows. A parasite is an organism that takes some elaborated resources from others living beings, and therefore the genetic programs of parasites, coding for restricted metabolic pathways, are shorter than those in wild ones. It has been found that the shorter programs of parasites are more competitive than entire and large ones of wild varieties:

"The mechanisms of feedback inhibition and/or gene repression which would help the parental strain to economize are not universal and/or not complete enough to offset the advantage that the "defective" strain has in not being involved at all in this particular biosynthetic process" (Zamenhof et al., 1962).

That parasites have shorter genomes was known since 1887, when Bovery discovered that Ascaris has in its somatic cells only a part of the chromatin which is present in the germ line cell nuclei (see Ammermann, 1985).

Viruses is another "easy case" for selection in favor of simplified genomes: their code has overlapping genes using different reading frames, overlapping genes using alternative promoters in the same reading frame and differential splicing of a single premessenger to produce a variety of polypeptides in adenovirus. Such a proficiency in information packing leads one to think that in a virus everything is fundamental. In this regard, the following comments that appear in Cavalier-Smith (1985), and that he attributes to Joklik, are very illustrative:

"Plasmid DNA and virus chromosomes are always much smaller than cellular chromosomes. As they lack significant amounts of non-coding DNA reasonably strong selective forces must exist to prevent its accumulation. For viruses the nature of this force is fairly obvious. The closed geometry of a viral capsid cannot contain more than a limited amount of DNA. Moreover capsid size, unlike cell

size, cannot increase gradually but would have to evolve in major discontinuous steps that are inherently improbable". By the same token, he proposes that "the uniformly low chloroplast genome size is a sign of strong selection against wasteful noncoding DNA".

Nonetheless, one cannot think that minimal descriptions are all to nature. Just consider that if the enzymes responsible for the destruction of poisons are produced in more abundance, poison neutralization is made faster. This can be achieved by gene amplification.

One might think that the problem of redundancy of the genome can be settled once and forever with experimental deletions. It seems that, in general, large deletions have a greater tendency to be lethal than large duplications (Cavalier-Smith, l.c. 1985). This seems to imply that the genome is highly simplified. Nevertheless, experimental deletions (of low size) has shown today that in *S. cerevisiae* that are cultivated in rich media less than 20% of genes are essential, and the large majority of the remaining deletions have little or no detectable effect on growth. Moreover, similar observations have been made for other eukaryotes and prokaryotes. Is this discovery a direct solution of the mystery of the pink strip saying that the genome is highly redundant? Not at all because a different explanation also must be studied: the genome is functionally redundant and moreover it has many genes that exist to be used under poor environments and/or extreme conditions that were not tested anyhow (Ihmels et al, 2007).

We have now a contrast: on one hand, we have evolution that is in literal sense open ended and whose fundamental mechanism could be *duplicate + adjust + delete*, but on the other, the very existence of species shows that evolution has got tired. How do we explain the fatigue of evolution when the possibilities of evolution are unbounded at any one moment?

Our answer is this: the cell is computing machinery that is always trying new ways but the problem is that it has not enough computing power to devise at once the highly functional code that is demanded to synthesize new functions that are acceptable for selection.

96. ***Challenge.*** *Make a program to calculate tables of critical values for the diverse distributions.*

97. ***Exercise.*** *Merge into a single program the code for p-values to that of critical values.*

98. ***Optional exercise.*** *Use the previous program to update your toolBox6 to version 7. Nevertheless, it might be preferable to make the actualization after checking that we can correctly plug the various services of this program to coming needs.*

4.5 Conclusion

We have developed software to calculate critical and p-values of the four fundamental distributions of statistical inference of the normal distribution, z, t, χ^2, F. Our precision seems to fit that of Gnumeric with the exception of the critical values of the χ^2 distribution. To correctly understand what happens is a challenge for everybody. We have faced the problem of simplifying repetitions in various programs that are merged into a single unity. We have adopted a professional solution which is based on abstract classes. Let us notice that abstract classes is an expedite way into high levels of variability.

Chapter 5

The dictionary of experiments

`The most famous 12 tests.`

Statistical inference has as aim to extrapolate conclusions based on random samples to propositions that are applied to the entire population. The problem of making that in a reasonable way is forced here to fit the next universal format: **science** is before everything else a contrast between what we see and what we think or believe. This general format is specialized in many ways to model specific experiments.

99. ***Purpose*** *We implement the statistical analysis of the twelve fundamental experiments of elementary science that are directly tied to the normal, binomial or multinomial distributions.*

5.1 The simplest experiment

Here we study the case in which one picks data, random data, to judge a proposition about the populational mean or to estimate its confidence interval. We need some notation that will remain valid in the sequel.

100. ***Notation.***

1. H_o the null hypothesis.
2. H_a the alternate hypothesis.
3. μ_X, μ_Y populational means of variables X and Y.
4. $\bar{X}_n$, mean of sample with n elements.
5. σ the deviation of the population. May have subindixes corresponding to random variables.

6. σ^2 the variance of the population.
7. $\widehat{\sigma_X}$, $\widehat{\sigma_X}^2$ sample deviation and variance.
8. p, f, g, populational and sample proportions.
9. $z_{(\alpha,2)}$ critical value of the z-distribution for significance α and two tails. Similar notations for the other distributions.

101. ***Study of a mean knowing*** σ

We use the mean of a sample knowing the population variance to judge the populational mean.

Two tails test: $H_o : \mu_x = 20$ and $H_a : \mu_x \neq 20$

One tail test: $H_o : \mu_x = 20$ and $H_a : \mu_x < 20$

Stadigraph to measure the discrepancy between what we see $\bar{X_n}$ and what we think or expect μ_X:

$$z = \frac{\bar{X_n} - \mu_X}{\frac{\sigma_X}{\sqrt{n}}}$$

If the discrepancy is found to surpass the critical value of the z distribution associated to the significance, it is recommended to reject the null hypothesis and to look for another explanation apart from that taken to explain the bulk of the distribution. If the discrepancy is less than critical value, one is invited to belief that observed facts are explained by the very same mechanisms that explain the bulk of the distribution.

Confidence interval for the mean μ:

$$\bar{X_n} - (z_{(\alpha,2)})\frac{\sigma_X}{\sqrt{n}} < \mu < \bar{X_n} + (z_{(\alpha,2)})\frac{\sigma_X}{\sqrt{n}}$$

Number of data necessary to asses the confidence interval with error E (half of the confidence interval):

$$n = ((z_{(\alpha,2)})\frac{\sigma_X}{E})^2.$$

102. ***The code for a z-test for a mean follows.***

```
//Program F102 zTestOneMean
//Runs a z-test for the mean
//knowing the mean of a random sample
//and the populational variance.
//Confidence interval is also calculated.
public class zTestOneMean
{

//Prints raw data
private static void printData(double xBar, double mu,
```

```
double sigma, double n,  double alpha)
{
 System.out.println("\nz Test to study the null" +
 "\nhypothesis about a mean  given the mean " +
 "\nof a random sample and the standard deviation" +
 "\nof the population.\n");
 System.out.println("xBar = " + xBar);
 System.out.println("mu = " + mu);
 System.out.println("sigma = " + sigma);
 System.out.println("n = " + n);
 System.out.println("alpha = " + alpha);
}



/*z Test to study the null hypothesis about a mean
 * given the mean of a sample and the standard deviation
 * of the population.*/
private static double  zHoOneMean(double xBar, double mu,
double sigma, double n)
 {
  System.out.println(
  "zExp = (xBar- mu)/( sigma/Math.sqrt(n))");
  double zExp = (xBar- mu)/( sigma/Math.sqrt(n));
  return zExp;
 }

/*Confidence interval for the mean
given the populational variance */
private static void intConfMediaZ(double xBar, double mu,
double sigma, double n,  double alpha)
{
 System.out.println( "\nConfidence interval of the mean" +
" \ngiven te populational deviation");
 double zCriticTwoTails = 2;
 double r = zCriticTwoTails*sigma / Math.sqrt(n);
 System.out.println( "r = z*sigma / Math.sqrt(n)" );
 System.out.printf( "r = %10.4f \n", r );
 double limInf = xBar - r;
 double limSup = xBar + r;
 System.out.printf( "limInf = xBar - r " +
                     "= %10.4f \n" , limInf );
 System.out.printf( "limSup  = xBar + r " +
                     "= %10.4f \n", limSup );
}
```

```
//Standard error of the mean
private static double sError(double sigma, double n)
{
 System.out.println("\nStandard error  = " +
"sigma/Math.sqrt(n)");
 return sigma/Math.sqrt(n);
}

//Needed n = size of the sample
//given tolerance E in the evaluation of the mean.
private static int nNeeded(double sigma, double E)
{
 System.out.println("\nTolerance error  E = "+ E);
 double zCriticTwoTails = 2;
 System.out.println("m= " + "zCriticTwoTails * sigma / E ");

 double m = zCriticTwoTails * sigma / E;
 System.out.println("Needed  n = m*m");
 //Upwards approximation
 return (int) Math.ceil(m*m);
}

/*Studies about a mean
 * given the mean of a sample and the standard deviation
 * of the population.*/
private static void zOneMean()
 {
  //Example 184, pg 169
  double xBar = 40;
  double mu = 20;
  double sigma = 16;
  double n = 36;
  double alpha = 0.05;
  printData(xBar,mu,sigma,n,alpha);
  double zCriticTwoTails = 2;
  double zCriticOneTail = 1.6;
  double zExp = zHoOneMean(xBar,mu,sigma,n);
  System.out.printf( "zExp = %10.4f \n" , zExp);
  System.out.printf("zCritic for TWO TAILS " +
"= +/-%10.4f \n",  zCriticTwoTails);
  System.out.printf("zCritic for ONE TAIL  " +
     "= +/-%10.4f \n",  zCriticOneTail);
  double error = sError(sigma,n);
  System.out.printf("Standard error  = %10.4f \n", error);
  intConfMediaZ(xBar,mu,sigma,n,alpha);
  //Admitted tolerance for mu
```

```
  double E = 5;
  int nNeeded = nNeeded(sigma,  E);
  System.out.printf("Needed n = %10d \n", nNeeded);
 }


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

103. ***Exercise.*** *Run the program and play with the code.*

104. ***Let us plug our z-test for a mean to the code about critical and p-values:***

```
//Program F104 zTestOneMean2
//Runs a z-test for the mean
//knowing the mean of a random sample
//and the populational variance.
//Confidence interval is  calculated.
//Critical and p-values are also calculated.
//Style = global variables.

public class zTestOneMean2
{

 //Outer class is instantiated
 static zTestOneMean2 a = new zTestOneMean2();

 //This is constant pi
 private static final double  pi = 3.14159265358979323846;

//Symbolic initialization that can be modified
 private static double alpha = 0.05;
 //Degrees of freedom of the t distribution
 private static double dfT = 9;
 //Degrees of freedom of the chi2 distribution
 private static double dfChi2 = 17;
 //Degrees of freedom of the F distribution, numerator
 private static double dfFNum = 4;
 //Degrees of freedom of the F distribution, denominator
 private static double dfFDen = 32;

//====== Methods of general use==========
```

```
//Returns numb!
 private static long factorial (long numb)
  {
   if (numb <=1 ) return 1;
  else
   return numb * factorial (numb -1);
  }

//Returns the double factorial
private static double doubleFact(double x)
{
if (x <=0 ) return 1;
else
 return x * doubleFact(x -2);
}


//Returns the Gamma function
private static double Gamma(double x)
{
 int xRounded = (int) x;
 //System.out.println( x + "   " + xRounded);
 if (xRounded == x)
return factorial (xRounded-1);
 else
  {
 return Math.sqrt(pi)* doubleFact(2*xRounded -1) /
          Math.pow(2,xRounded);
  }
}



//======Inner top class with commonalities=========
//The methods that eventually can be useful
//to other distributions come here such as they are.
//The methods that are privative of each distribution
//are defined here as abstract
//and implemented in  specific classes below.
abstract class ValueAbstract
{
 double sumzero;
 double Error;
 double jCritic;
 double k;
```

```
 //The density function is of abstract type.
 //It is not implemented,
 //so we can implement other functions too and
 //process them with this very same machinery.
 abstract double f(double t);



//========= p-values==========

//Returns the integral under the function f
//in within 0 and z. Precision=1/N.
//Simpson's rule adapted for reuse.
double HalfBodySimpson(double z, long N)
 {
  double h = z/(2*N);
  long m = N;
  double sum = f(0);
  for(int j= 1; j<=m; j++)
 sum = sum +   4* f((2*j-1) * h);
  for(int j= 1; j<m; j++)
 sum = sum +   2* f((2*j) * h);
  sum = sum + f((2*m) * h);
  sum = (h/3)*sum;
  return sum;
 }


//The area in within zero and z of chosen
//distribution is calculated
//for various degrees of precision.
double area(double z)
 {
  z = (Math.sqrt(1+4*z*z)-1)/(2*z);
  int N;
  double  answer;
  int i = 3;
  N = (int) Math.pow(10, i);
  answer = HalfBodySimpson(z,N);
  return answer;
 }

//Returns Psi(e) = p( x < e)
```

```
//the cumulative distribution function of e.
//This is abstract because it depends
//on the distribution, it must be
//implemented in the corresponding classes.
double psi(double e)
{
return 0.5 + area(e);
}

//Returns the p-value(e) = p( x > e)
//Local implementation of the p-value of  e
double pValue(double e)
{
 return 1- psi(e);
}


//Local manager
void pExecutor(double testValue)
{
 System.out.println("\nTest value = " + testValue);
 System.out.print("Simpson's rule."+
  "\nArea = p( 0 <  x < " + testValue + ") = ");
 double area =  area(testValue);
 System.out.println( area);
 //psi(e) = p( x < e)
 double psi = psi(testValue);
 System.out.println("psi(" + testValue + ") " +
    "= p( x < " + testValue + ") = " +  psi );
 double pValue = pValue(testValue);
 System.out.println("p-value(" +
      testValue + ") = " +
    "p( x > " + testValue + ") = " + pValue);
}

//========Critical values=====

//Reports the transformed upper u-critical for
//the given distribution.
//Significance level = alpha.
//Precision=1/N.
//Simpson's rule adapted for reuse.
//The error of approximation is also calculated.
double upperU(double h, double target)
{
```

```
double r = h/6;
jCritic = 0;
double sum = sumzero;
for(int j= 0; sum < target; j++)
{
 sum = sum +
 r*(f( j*h) + 4* f((j+0.5) * h) +  f((j+1) * h));
 jCritic = j+1;
}
jCritic = jCritic-1;
double greatSum = sum -
      r*(f( jCritic*h) + 4* f((jCritic+0.5) * h)
         +  f((jCritic+1) * h));
Error = target-greatSum;
double uAprox = (jCritic) *h;
return uAprox;
}



//Returns the transformed value z = u/(1-u*u)
double z(double u)
{
double z = u/(1-u*u);
return z;
}

//The wide of subintervals is defined
double h()
{
double u = 1;
int i=3;
long N = (int) Math.pow(10, i);
double h = u/N;
return h;
}

//Reports the upper z-critical for
//Significance level = alpha.
double vCrit(double target)
{
double h = h();
//System.out.println( "h = " + h);
double uCritic =  upperU(h, target);
double zCritic = z(uCritic);
return zCritic;
```

```
}


//*****Analytical hunting*******

//Returns the density function of the distribution
abstract double ff(double z);



//Returns F(a,b) = h(f(a)  + 4 * f((a+b)/2)  + f(b))-Error;
double F(double a, double b)
{
double h = (b-a)/6;
double  Fb = h* (  ff(a) + 4 * ff((a+b)/2)
                 + ff(b)  )-Error;
return Fb;
}


//Returns the  ratio  F(zCritic,gi)/ ff(gi)
double ratio(double zCritic, double gi)
{
double  ratio =  F(zCritic,gi)/ ff(gi) ;
return ratio;
}

//Critical value for the upper tail.
//A professional  approximation is found by
//analytical hunting
double hunting(double zCritic )
{
double h = h();
double ur =  (jCritic+1)*h;
//Right border of the critical interval
double rBorder = z(ur);
double gSubi = rBorder;
double gSubiPlus= 0;
int iterations = 0;
int maxIterations = 30;
boolean more = true;
while (more)
{
gSubiPlus = gSubi -  ratio(zCritic, gSubi);
iterations = iterations +1;
if (iterations >= maxIterations)  more = false;
```

```
if (gSubiPlus == gSubi) more = false;
gSubi = gSubiPlus;
}
return gSubiPlus;
}

//Critical value for significance level alpha, one tail
double  critical(  double alpha)
{
double target = 1-alpha;
double vCritic = vCrit(target);
vCritic = hunting(vCritic);
return vCritic;

}

//Local manager
void critManager(double alpha)
{
System.out.println( "\nAlpha = Significance level = "
+ alpha + ".");

System.out.println("Critical values for ONE TAIL = +/-"
+  critical(alpha) );

double alphaHalf = alpha/2;
System.out.println("Critical values for TWO TAILS = +/-"
            + critical(alphaHalf) );
}

}//end of class pValueAbstract


//======Particular classes========


//The z class.
//Definition of the z density function
//and exclusive to it methods.
//It is an specification of the ValueAbstract class.
class VzClass extends ValueAbstract
{

 //Constructor: this is the first thing to do
 VzClass()
 {
```

```
  sumzero = 0.5; //half bell
  System.out.println("\n===== z distribution");
 }


//Function f is implemented here.
//Returns the transformed density function of the
//standardized normal distribution.
double f(double t)
{
 double z = t/(1-t*t);
 double  f = 1 /Math.pow(2*pi, 0.5)*Math.exp(-z*z/2)*
             (1+ t*t)/( (1-t*t)*(1-t*t));
 return f;
}


//Returns the density function of the Z distribution
double ff(double z)
{
 double  f =  Math.exp(-z*z/2)/Math.pow(2*pi, 0.5);
 return f;
}

}//end of class VzClass


//The t class
//Definition of the t density function
//and exclusive to it methods.
//It is an specification of the ValueAbstract class.
class VtClass extends ValueAbstract
{

//Constructor: this is the first thing to do
VtClass()
{
 sumzero = 0.5; //half bell
 System.out.println("\n===== t distribution");
 System.out.println("t degrees of freedom = " + dfT);
}



//Function f is implemented here.
```

```
//Returns the transformed density function of the
//t distribution with dfT d.f.
double f(double t)
{
 double nu = dfT;
 double z = t/(1-t*t);
 double  f = ( Gamma((nu+1)/2) /
       ( Math.sqrt(pi*nu)*Gamma(nu/2)) ) *
 Math.pow(1+z*z/nu, -(nu+1)/2 )  * (1+ t*t)/( (1-t*t)*(1-t*t));
 return f;
}


//Returns the density function of the t-distribution
double ff(double z)
{
double nu = dfT;
double  f =  ( Gamma((nu+1)/2) /
       ( Math.sqrt(pi*nu)*Gamma(nu/2)) ) *
    Math.pow(1+z*z/nu, -(nu+1)/2 ) ;
return f;
}

}//end of class VtClass



//The chi2 class
//Definition of the Chi2 density function
//and exclusive to it methods.
//It is an specification of the ValueAbstract class.
class VChi2Class extends ValueAbstract
{

//Constructor: this is the first thing to do
VChi2Class()
{

 sumzero = 0; //no negative values
 System.out.println("\n===== Chi2 distribution");
 System.out.println("Chi2 degrees of freedom = " + dfChi2);
}

//Returns Psi(e) = p( x < e)
//the cumulative distribution function of e
double psi(double e)
```

```
{
 return area(e);
}

//Function f is implemented here.
//Returns the transformed density function of the
//chi2 distribution with nu d.f.
double f(double t)
{
 double nu = dfChi2;
 double z = t/(1-t*t);
 double  f = ( 1 /
   ( Math.pow(2, nu/2)*Gamma(nu/2)) )
   * Math.exp(-z/2)* Math.pow(z, (nu-2)/2)
   * (1+ t*t)/( (1-t*t)*(1-t*t));
 return f;
}



//Returns the density function of the chi2-distribution
double ff(double z)
{
double nu = dfChi2;
double  f = (1/(Math.pow(2, nu/2)*Gamma(nu/2)))
 * Math.exp(-z/2)* Math.pow(z, (nu-2)/2);
return f;
}


//Local manager
void manager(double alpha)
{
System.out.println("Critical values for ONE TAIL "
    + "\nof the chi-square- "
+  "distribution with df =  "
+ dfChi2
+ "\nAlpha = Significance level = "
+ alpha + ".");

System.out.println("Lower critical value  = "
+  critical(1-alpha) );
System.out.println("Upper critical value  = "
+  critical(alpha) );
System.out.println("\nCritical values for TWO TAILS "
    + "\nof the chi-square "
```

```
+  "distribution with df=  "
+ dfChi2
+ "\nAlpha = Significance level = "
+ alpha + ".");
double alphaHalf = alpha/2;
double Chi2Menos = critical(1-alphaHalf) ;
System.out.println("Lower critical value  = "
+  Chi2Menos  );
double Chi2Mas = critical(alphaHalf) ;
System.out.println("Upper critical value  = "
+  Chi2Mas  );
}
}//end of class pVChi2Class



//The F class
//Definition of the Chi2 density function
//and exclusive to it methods.
//It is an specification of the ValueAbstract class.
class VFClass extends ValueAbstract
{

//Constructor: this is the first thing to do
VFClass()
{
sumzero = 0; //no negative values
double nu1 = dfFNum;
double nu2 = dfFDen;
double a =  Math.pow(nu1, nu1/2);
double b =  Math.pow(nu2, nu2/2);
double c = beta(nu1/2,nu2/2);
k = a* b / c;
System.out.println("\n===== F distribution");
System.out.println("F degrees of freedom: "
           + "Num = " + dfFNum
           + ", Den = " + dfFDen);
}

//Returns Psi(e) = p( x < e)
//the cumulative distribution function of e
double psi(double e)
{
return area(e);
}
```

```
//Returns the transformed density function of the
//F distribution with dfFNum and dfFDen d.f.
double f(double t)
{
double nu1 = dfFNum;
double nu2 = dfFDen;
double z = t/(1-t*t);
double  f =   k*Math.pow(z, nu1/2-1)
                  * Math.pow(nu2+nu1*z, -(nu1+nu2)/2)
            * (  (1+ t*t)/( (1-t*t)*(1-t*t))  ) ;
//System.out.println(f);
return f;
}


//Returns a helping  function  with   d.f. =
//a(num) and b(den)
double g(double x, double a, double b)
{
double  g = Math.pow(x, a -1)*
                (Math.pow(1-x, b -1));
return g;
}


//Returns the integral under the function f
//in within 0 and u. Precision=1/N.
//Simpson's rule adapted for reuse.
//a and b are degrees of freedom
double HalfBodySimpsong(double u,
double a, double b, long N)
{
double h = u/(2*N);
long m = N;
double sum = g(0,a,b);
for(int j= 1; j<=m; j++)
 sum = sum +   4* g((2*j-1) * h,a,b);
for(int j= 1; j<m; j++)
 sum = sum +   2* g((2*j) * h,a,b);
sum = sum + g((2*m) * h,a,b);
sum = (h/3)*sum;
return sum;
}
```

```
//Returns the  beta  function in integral form
//with a and b d.f.
double beta( double a, double b)
{
double  answer;
int i = 4;
int  N = (int) Math.pow(10, i);
double u=1;
answer = HalfBodySimpsong(u,a,b,N);
return answer;
}


//Reports the transformed upper u-critical for
//the given distribution.
//Significance level = alpha.
//Precision=1/N.
//Simpson's rule adapted for reuse.
//The error of approximation is also calculated.
double upperU(double h, double target)
{

double r = h/6;
double sum = 0;
jCritic = 0;
for(int j= 0; sum < target; j++)
{
sum = sum +
  r*(f( j*h) + 4* f((j+0.5) * h) +  f((j+1) * h));
jCritic = j+1;
}
jCritic = jCritic-1;
double greatSum = sum -
          r*(f( jCritic*h) + 4* f((jCritic+0.5) * h)
        +  f((jCritic+1) * h));
Error = target-greatSum;
double uAprox = (jCritic) *h;
return uAprox;
}


//Returns the density function of the t-distribution
double ff(double z)
{
double nu1 = dfFNum;
```

```
double nu2 = dfFDen;
double  f = k*Math.pow(z, nu1/2-1)
 * Math.pow(nu2+nu1*z, -(nu1+nu2)/2)   ;
return f;
}

//Local manager
void manager(double alpha )
{
  System.out.println("\nF distribution");
System.out.println("Critical values for ONE TAIL "
    + "\nof the F- "
+ "distribution with df:  "
+ dfFNum + " and " + dfFDen+ ", "
+ "\nAlpha = Significance level = "
+ alpha + ".");

System.out.println("Lower critical value  = "
+  critical(1-alpha) );
System.out.println("Upper critical value  = "
+  critical(alpha) );

System.out.println("\nCritical values for TWO TAILS "
    + "\nof the f  "
+ "distribution."
+ "\nAlpha = Significance level = "
+ alpha + ".");
double alphaHalf = alpha/2;

System.out.println("Lower critical value  = "
+  critical(1-alphaHalf) );
System.out.println("Upper critical value  = "
+  critical(alphaHalf) );
}
}//end of class pVFClass


//=====================================
//======== THE 10 MOST USUAL TESTS ======
//=====================================


//======= z test for one mean ===========

//Program F102 zTestOneMean
//Runs a z-test for the mean
```

```
//knowing the mean of a random sample
//and the populational variance.
//Confidence interval is also calculated.
public class zTestOneMean
{


void printData(double xBar, double mu,
double sigma, double n,  double alpha)
{
System.out.println("\nz Test to study the null" +
 "\nhypothesis about a mean  given the mean " +
 "\nof a random sample and the standard deviation" +
 "\nof the population.\n");
System.out.println("xBar = " + xBar);
System.out.println("mu = " + mu);
System.out.println("sigma = " + sigma);
System.out.println("n = " + n);
System.out.println("alpha = " + alpha);
}


/*z Test to study the null hypothesis about a mean
* given the mean of a sample and the standard deviation
* of the population.*/
double  zHoOneMean(double xBar, double mu,
double sigma, double n)
{
System.out.println(
  "zExp = (xBar- mu)/( sigma/Math.sqrt(n))");
double zExp = (xBar- mu)/( sigma/Math.sqrt(n));
return zExp;
}

/*Confidence interval for the mean
given the populational variance */
void intConfMediaZ(double xBar, double mu,
double sigma, double n,  double alpha)
{
System.out.println( "\nConfidence interval of the mean" +
" \ngiven the populational deviation");
double zCriticTwoTails = 2;
double r = zCriticTwoTails*sigma / Math.sqrt(n);
System.out.println( "r = z*sigma / Math.sqrt(n)" );
System.out.printf( "r = %10.4f \n", r );
double limInf = xBar - r;
```

```
double limSup = xBar + r;
System.out.printf( "limInf = xBar - r " +
                      "= %10.4f \n" , limInf );
System.out.printf( "limSup  = xBar + r " +
                      "= %10.4f \n", limSup );
}

//Standard error of the mean
double sError(double sigma, double n)
{
System.out.println("\nStandard error  = " +
"sigma/Math.sqrt(n)");
return sigma/Math.sqrt(n);
}

//Needed n = size of the sample
//given tolerance E in the evaluation of the mean.
int nNeeded(double sigma, double E)
{
System.out.println("\nTolerance error  E = "+ E);
double zCriticTwoTails = 2;
System.out.println("m= " + "zCriticTwoTails * sigma / E ");

double m = zCriticTwoTails * sigma / E;
System.out.println("Needed  n = m*m");
//Upwards approximation
return (int) Math.ceil(m*m);
}

/*Studies about a mean
* given the mean of a sample and the standard deviation
* of the population.*/
void zOneMean()
{

}

}//End of zTestOneMean


//General manager:

public static void ztestOneMean( )
{
 //Example 184, pg 169
 double xBar = 40;
```

```
 double mu = 20;
 double sigma = 16;
 double n = 36;
 alpha = 0.05;
 //Top class is declared
 ValueAbstract p;
 //Particular class z is instantiated
 VzClass z = a.new VzClass();
 //Top class is instructed to take
 //the form of the particular class
 p = z;

 //======z-test for the mean ========
 zTestOneMean b = a.new zTestOneMean();

 b.printData(xBar,mu,sigma,n,alpha);
 //Critical values
 double zCriticOneTail = p.critical(alpha);
 double alphaHalf = alpha/2;
 double zCriticTwoTails = p.critical(alphaHalf);
 //z-test
 double zExp = b.zHoOneMean(xBar,mu,sigma,n);
 System.out.printf( "zExp = %10.4f \n" , zExp);
 double pValue = p.pValue(zExp);
 System.out.println("p-value(" +
      zExp + ") = " +
    "p( x > " + zExp + ") = " + pValue);
 System.out.printf("zCritic for TWO TAILS " +
  "= +/-%10.4f \n",  zCriticTwoTails);
 System.out.printf("zCritic for ONE TAIL  " +
    "= +/-%10.4f \n",  zCriticOneTail);
 //Standard error
 double error = b.sError(sigma,n);
 System.out.printf("Standard error  = %10.4f \n", error);
 //Confidence interval
 b.intConfMediaZ(xBar,mu,sigma,n,alpha);
 //Admitted tolerance for mu
 double E = 5;
 //Size of sample to attain tolerance E
 int nNeeded = b.nNeeded(sigma,  E);
 System.out.printf("Needed n = %10d \n", nNeeded);
}

public static void main(String args[])
{
 ztestOneMean();
```

```
}
}//End of main class
```

105. ***Exercise.*** *Run the program and play with the code.*

106. ***Let us update our toolBox6 to version 7.***

```
//Program F106 toolBox7
/*
 This is a tool box of methods to calculate
  a) some statistics of a list of data,
  b) the pstricks code for a bar diagram,
  c) the binomial distribution and its critical values,
  d) a z-test for one mean.

 How to use this program:

 1. Append it to any project and run it.
 2. This program is a tool box because its main method
    can be filled with any allowed method taken from the
    demo.
 3. Most relevant allowed methods can be listed in the main
    but only one must be activated, all others must be
    muted.
 4. To silence one method, append // as a prefix.
    To activate a method, delete the prefix //.
 5. For a given task, choose the best suited method
    among the diverse variants.
 6. To run a method with your own data,
    modify the inbuilt data according to your needs.
 7. This first version has just one aim:
    To implement fundamental tasks as transparently
    as possible. So, it looks rather like an array of
    independent, non interacting  methods.
 8. To inquire about unknown terms, drag the mouse over
    the term and wait until Eclipse displays the
    corresponding documentation.
 9. To go   from the name of a
    method  to its implementation:
     Pose the cursor at the end of the line
     with the name of the method and press the space bar,
     next press F3: you will be transported to the
     implementation of the method.
 10. To mark a specific site in the program:
     -Activate line numbering.
     -Pose the cursor over the line you want to mark.
```

```
     -Go to the corresponding number at the left margin
      of the Editor Window and right click.
     -Choose the Add Bookmark menu and punch it.
     -A blue marker will appear in the right margin of the
      Editor Window. By clicking over it, one can
      return to the desired place.
      One can also play with the menu
      Window -> ShowView -> Outline.
      A list with all methods will appear:
      click over any one and you will
      transported to the corresponding place
      along the text of the program.

 We follow our text in
 Elementary Statistics Vol I version 7.
*/



public class toolBox7
{

 //The outer application is instantiated
 private static toolBox7 a = new toolBox7();
 //Top abstract class is declared
 private static  ValueAbstract p;
 //Particular class z is instantiated
 private static VzClass z =  a.new VzClass();


//============================
//MAIN--MAIN--MAIN--MAIN--MAIN
//============================

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

 //Demo
 public static void demo()
  {
   meanVarList();
   meanVarFreqTable();
   meanVarCumTable();
   groupData();
```

```
   codePSTricks();
   binomialDistribution();
   zTestOneMean();
  }

//=============================================
//               IMPLEMENTATIONS
//=============================================




//=============================================
//      Methods of general use
//=============================================

//Prints a separator
private static void separator()
 {
  System.out.println( "\n========*=========\n");
 }


//Prints a list and its title
private static void printList(double DataVector[],
                              String title)
{
 System.out.println( "\n " + title );
 int nData = DataVector.length;
 boolean go = true;
 int nCols = 10;
 int nRows = 9;
 int counter = 0;
 for (int i = 0; (i < nRows) & go; i++)
  {
   for (int j = 0; (j < nCols) & go; j++)
    {
     System.out.print( DataVector[i*10 + j] + "   ");
     if (counter >= nData-1) go = false;
     counter = counter+1;
    }
  System.out.println( );
 }
 System.out.print("\nNumber of data = " +nData + "\n");
}
```

```
//Prints n data in vector
private static void printVector(double Vect[], int n)
{
 //System.out.println( "Data are" );
 for ( int i = 0; i < n; i++ )
   System.out.println(i + "\t" +  Vect[ i ]  );
}


//Sums data in vector with real numbers
private static double sumVector(double Vect[], int n)
{
 double sum = 0;
 for ( int i = 0; i < n; i++ )
 sum = sum + Vect[i];
 return sum;
}

//Prints  a matrix and is title
private static void printMatrix(double Matrix[][],
                        String title)
{
 System.out.println( title );
int m = Matrix.length;
 for ( int i = 0; i < m; i++ )
 System.out.println( Matrix[ i ][0]+
        "\t "+ Matrix[ i ][1]);
}


//Documentation is printed
private static void explanation()
{
 System.out.println( "\nCoeff around 0.01, " +
                     "homogeneous population." +
                     "\nCoeff around 0.1, " +
                     " +/- homogeneous population."+
                     "\nCoeff around 0.5, " +
                     "  heterogenous population.");
}


//===============================================
//       MEAN AND VAR O A LIST OF DATA
//===============================================
```

```
//======== Inner class definition=======

//This inner class defines a new type: Table.
//It converts array F (two dimensional) into
//a class or object, ready for reuse.
private  class Table
{
 //The class Table fundamentally has three elements:
 //a matrix with two columns and its length,
 //which is the number of rows and
 //the titles of columns.
 int length;
 String[] Titles = { "x",  "x2 = x^2"};
 double M[][] = new double[2000][2];

 //The input of Table is defined here
 //by means of an initialization procedure
 //that is called constructor.
 //The natural input is the raw data,
 //which is consigned in a one dimensional array.
 Table(double[] V )
 {
  length = V.length;
  //The matrix M of Table is initialized
  //Raw data go in the first column.
  //Squares in the second.
  for(int i = 0; i < length; i++)
  {
M[i][0] = V[i];
M[i][1] = V[i]*V[i];
  }
 }

 //Data of column k are summed.
 //The input is a table, the output is of type double.
 private  double sum(Table  T, int k)
 {
  double sum = 0;
  for(int i = 0; i < T.length; i++)
 sum  = sum +  T.M[i][k];
  return sum;
 }
```

```
 private  void printTable(Table T)
  {
   System.out.printf("%n %10s  %11s",
              T.Titles [0], T.Titles [1]);
   for ( int i = 0; i < T.length; i++ )
  System.out.printf("%n %10.3f   %10.3f ",
        T.M[i][0], T.M[i][1] );
   System.out.println( );
  }
}//end of inner class


//=== Methods of the outer class ===


//This is the core of the main class
//Here, commands are given to execute the overall task.
private static void meanVarVector(double[] V)
{
 //A table is declared
 Table T =  a.new   Table(V);
 T.printTable(T);
 double sumx = T.sum(T,0);
 int n = T.length;
 double mean = sumx/ n;
 double sumx2 = T.sum(T,1);
 System.out.printf("%n Sum %6.3f   %10.3f ",
                sumx,        sumx2 );
 System.out.print("\n \n");
 System.out.println("n = number of data = " + n);

 System.out.println("\nsumx = " + sumx);
 System.out.println("Mean = sumx / n");
 System.out.println("Mean = " + mean);
 System.out.println("sumx2 = " + sumx2);
 System.out.println("variance = " +
   "(n * sumx2 - sumx*sumx)/ (n*( n-1))");
 double var = (n * sumx2 - sumx*sumx)/ (n*( n-1));
 System.out.println( "Variance = "+ var);
 double deviation = Math.pow( var, 0.5);
 System.out.println( "Deviation = "+ deviation);
 double coeffVar =  deviation/mean;
 System.out.println( "Coefficient of variation " +
                 "= deviation/mean");
 System.out.println( "Coefficient of variation = "
                                  + coeffVar);
```

```
 explanation();
}

//A list of data  is  processed
private static void meanVarList( )
{
  System.out.println("\n\n");
  System.out.println( "MEAN AND VAR OF A LIST OF DATA");
  //Example 19 page 9.
  double  List[]  = {5, 3, 4.5, 2};
  //Prints list and title
  printList(List,  "Raw data");
  System.out.println( "\nTABLE WITH x AND ITS SQUARE");
  meanVarVector(List);
  separator();
}

//=================================================
// MEAN AND VAR OF A TABLE OF ABSOLUTE FREQUENCIES
//=================================================


//======== Inner class definition=======

//This inner class defines a new type: Table.
//It converts array F (two dimensional) into
//a class or object, ready for reuse.
private  class TableFreq
 {
//The class Table has two main elements:
//1) The titles of  the columns, whose length
//gives the number of columns.
//2) A matrix, whose length is the number of rows.

String[] Titles = { "x",  "F",   "xF", "x2F"};
double M[][] = new double[2000][Titles.length];
int nRows;
//The input of Table is defined here
//by means of an initialization procedure
//that is called constructor.
//The natural input is the frequency table
//which is consigned in a matrix FT with 2 columns.
TableFreq(double[][] FT )
 {
   nRows = FT.length;
   //The matrix M of Table is initialized
```

```
   //Raw data go in   columns 0 and 1.
   //Squares in the second.
   for(int i = 0; i < nRows; i++)
   {
 M[i][0] = FT[i][0];
 M[i][1] = FT[i][1];
 M[i][2] = M[i][0] * M[i][1];
 M[i][3] = M[i][0] * M[i][2];
   }
  }

//Data of column k are summed.
//The input is a table, the output is of type double.
private  double sum(TableFreq  T, int k)
 {
  double sum = 0;
  for(int i = 0; i < T.nRows; i++)
  sum  = sum +  T.M[i][k];
  return sum;
 }

private  void printTable(TableFreq T)
 {
  System.out.println("\nExtended matrix to " +
                "calculate variance:");
  System.out.printf("%n %10s %10s %10s %10s",
            T.Titles [0], T.Titles [1],
             T.Titles [2],T.Titles [3]);
    for ( int i = 0; i < T.nRows; i++ )
 System.out.printf(
"%n %10.3f %10.3f %10.3f %10.3f ",
T.M[i][0], T.M[i][1], T.M[i][2], T.M[i][3] );
  System.out.println( );
 }
}//end of inner class


//=== Methods of the outer class ===



// Mean and variance of a frequency table
private static void meanVarFreqTable(double FreqTable[][])
{
 //A table is declared
 TableFreq T =  a.new   TableFreq(FreqTable);
```

```
 T.printTable(T);
 double n = T.sum(T,1);
 double sumxF = T.sum(T,2);
 double sumx2F = T.sum(T,3);
 System.out.printf("Sum %18.3f %10.3f %10.3f ",
             n,   sumxF,        sumx2F );
 double mean = sumxF/ n;
 System.out.print(
  "\n\n n = Number of  objects = sum F = "+ (int) n);
 System.out.printf( "\n sum xF  = %10.3f ", sumxF);
 System.out.printf( "\n sum x2F  = %10.3f ",  sumx2F);
 System.out.printf( "\n Mean = sum xF / n = %10.3f ", mean);
 double Sxx  = sumx2F -    Math.pow(sumxF, 2)/n;
 System.out.printf(
 "\n Sxx = sum x2F - (sumxF)^2 /n = %10.3f ", Sxx);
 double var = Sxx / (n-1);
 System.out.printf(
 "\n Variance =  Sxx/(n-1) = %10.3f ", var);
 double deviation = Math.sqrt(var);
 System.out.printf(
 "\n Deviation =  %10.3f ", deviation);
 double c = deviation/ mean;
 System.out.printf(
   "\n Coefficient of variation =  %10.3f \n", c);
 explanation();
}


//A Frequency Table is processed
private static void meanVarFreqTable()
{
 System.out.println("\n\n");
 System.out.println("\n\nMEAN-VAR OF A FREQUENCY TABLE");
 //Frequency table: first coordinate = class marker
 //second coordinate = frequency

 //Example 20, pag 10
 double  FreqT[][] =
        {{4, 3},
         {7, 6},
         {10, 10},
         {13, 15},
         {16, 4},
         {19, 2}};

 printMatrix(FreqT, "\n Raw data");
```

```
 meanVarFreqTable(FreqT);
 separator();
}


//=================================================
//MEAN AND VAR OF A TABLE OF CUMULATIVE FREQUENCIES
//=================================================


private static void meanVarCumTable(double[][] cumFreqT)
{
 printMatrix(cumFreqT, "\n Raw data");
 //Absolute frequencies are calculated
 //and kept in FreqT, a matrix with two columns.
 double FreqT[][] = new double[cumFreqT.length][2];

 FreqT[0] = cumFreqT[0];
 int lx = cumFreqT.length;
 for(int i = 1; i < lx;  i++)
 {
  FreqT[i][0] = cumFreqT[i][0];
  FreqT[i][1] = cumFreqT[i][1]-cumFreqT[i-1][1];
 }
 System.out.println("\n\n");
 System.out.println(
 "\n MEAN-VAR OF ASSOCIATED  FREQUENCY TABLE");
  meanVarFreqTable(FreqT);//reuse
 }

//A cumulative frequency table is processed
public static void meanVarCumTable()
{
 System.out.println(
 "MEAN-VAR OF A CUMULATIVE FREQUENCY TABLE");
 //Cumulative table: first coordinate = class marker
 //second coordinate = cumulative frequency

 //Problem 1 of version 1,  pag 42.
 double  cumFreqT[][] =
        {{32,7},
         {40,12},
         {45,17},
         {50,45},
         {55,52},
         {60,58}};
```

```
  meanVarCumTable(cumFreqT);
  separator();
}


//=================================================
// GROUPING OF DATA IN INTERVALS + MEAN & VAR
//=================================================



//List is the input, a frequency table is the output
private static double[][] group(double List[],
        double Borders[], double classMarkers[]  )
{
 int nClasses = classMarkers.length;
 double FreqTable[][] = new double[nClasses][2];
 int nData = List.length;

 for(int j=0; j < nClasses; j++)
  {
FreqTable[j][0] = classMarkers[j];
FreqTable[j][1] = 0;
  }
 for(int i=0; i< nData; i++)
  {
   for(int j=0; j < nClasses; j++)
 {
  if((Borders[j] <= List[i]) & (List[i] < Borders[j+1]))
  FreqTable[j][1] =FreqTable[j][1] +1;
}

 }

 return FreqTable;
}




//Data is grouped into a frequency table.
//The borders and class marker of each interval
//is calculated.
//Final output = Mean and var of the freq table.
private static void group(double List[],
```

```
double infLimit, double intervalLength)
{
  //Calculate minimal value of data
  double min = List[0];
  for(int j=1; j < List.length; j++)
   if (List[j] < min)  min = List[j];
  //Calculate maximal value of data
  double max = List[0];
  for(int j=1; j < List.length; j++)
    if (List[j] > max)  max = List[j];
  System.out.println( "\nMax value = " + max);
  //Calculates number of classes
  int nClasses  = (int) ((max- infLimit)/intervalLength)+1;
  System.out.println( "Number of classes = " + nClasses);
  //Calculates interval borders and class markers
  //Grouping mode 2: defines inferior limit and length
  //of intervals.
  double classMarkers[] = new double[nClasses];
  double  Borders[] = new double[nClasses+2];
  Borders[0] = infLimit;
  classMarkers[0] = infLimit + intervalLength/2;
  for(int j=1; j < nClasses; j++)
    classMarkers[j] = classMarkers[j-1] + intervalLength;
  for(int j=1; j <= nClasses+1; j++)
    Borders[j] = Borders[j-1] + intervalLength;
  System.out.println( "\nBorders are" );
  printVector(Borders,nClasses+1 );
  System.out.println( "\nClass markers are" );
  printVector(classMarkers,nClasses );
 double FreqTable[][] = new double[nClasses][2];
 System.out.println(
 "\nEXTENDED TABLE OF ABSOLUTE FREQUENCIES" );
 //reuse
 FreqTable = group(List, Borders, classMarkers );
 meanVarFreqTable(FreqTable);
}


public static void groupM1()
{
//Declaration of data:
//observed values as a list
//Example 24 pag 12
double    List[] =
{
48.5, 49.2, 51.0, 50.9, 48.7, 50.5, 49.5, 50.8, 50.0, 51.1,
```

```
50.7, 51.1, 49.3, 49.1, 50.0, 48.7, 51.2, 49.2, 49.0, 49.3,
49.1, 48.7, 48.6, 49.2, 49.9, 50.1, 50.1, 50.9, 52.4, 50.2,
49.5, 50.9, 51.4, 49.7, 49.8, 50.8, 50.3, 51.8, 50.0, 51.1,
51.3, 50.1, 50.4, 51.1, 49.8, 49.8, 50.1, 50.7, 50.1, 49.9,
49.7, 51.1, 49.7, 49.9, 49.8, 50.5, 49.7, 50.8, 50.7, 50.7,
50.8,  51.5, 51.0
};
 System.out.println("\n\n");
 System.out.println("FROM DATA LIST TO FREQUENCY TABLE:");
 printList(List, "Raw data");
 //Borders of the intervals
 double  Borders[] = {48.5, 49.5, 50.5, 51.5, 52.5};
 double classMarkers[] = {49, 50, 51, 52};
 int nClasses = classMarkers.length;
 double FreqTable[][] = new double[nClasses][2];
 printList(Borders, "Borders of intervals");
 printList(classMarkers, "Class markers");
 FreqTable = group(List, Borders, classMarkers );
 meanVarFreqTable(FreqTable);
}



public static void groupM2( )
{
 //Declaration of data:
 //observed values as a list
 //Example 24 pag 12
 double    List[] =
{
48.5, 49.2, 51.0, 50.9, 48.7, 50.5, 49.5, 50.8, 50.0, 51.1,
50.7, 51.1, 49.3, 49.1, 50.0, 48.7, 51.2, 49.2, 49.0, 49.3,
49.1, 48.7, 48.6, 49.2, 49.9, 50.1, 50.1, 50.9, 52.4, 50.2,
49.5, 50.9, 51.4, 49.7, 49.8, 50.8, 50.3, 51.8, 50.0, 51.1,
51.3, 50.1, 50.4, 51.1, 49.8, 49.8, 50.1, 50.7, 50.1, 49.9,
49.7, 51.1, 49.7, 49.9, 49.8, 50.5, 49.7, 50.8, 50.7, 50.7,
50.8,  51.5, 51.0
   };
 System.out.println("\n\n");
 System.out.println("FROM DATA LIST TO FREQUENCY TABLE:");
 printList(List, "Raw data");

 //Calculate minimal value of data
 double min = List[0];
 for(int j=1; j < List.length; j++)
    if (List[j] < min)  min = List[j];
```

```
 System.out.println( "\nMin value = " + min);


 double infLimit = min;
 //Interval length
 double intervalLength = 1;
 System.out.println("Inferior limit = " + infLimit);
 System.out.println("Interval length = "
  + intervalLength);
 group(List, infLimit, intervalLength);
}

//Data is grouped and processed
public static void groupData( )
{
 //Method 1: all borders and class markers
 // are given by hand.
 groupM1();
 //Method 1: inferior border of first interval and
 //the length  of intervals are given.
 groupM2();
 separator();
}




//****************************
//Pstricks code
//****************************


//Produces the pstricks  code for the bar graph
//of a frequency table
public static void makeCode(double FreqTable[][])
{
 System.out.println("\nPSTRICKS code for the FreqTable\n");
 int n = FreqTable.length;
 //Width of each column
 double deltaX = 4;
 //The minimum x-coordinate is found
 double Minxx = 1000;
 for ( int i = 0; i < n; i++ )
  if (FreqTable[i][0] < Minxx)
        Minxx =   FreqTable[i][0];
 int Minxxx = (int) (deltaX*Minxx - deltaX);
```

```
 //The maximum x-coordinate is found
 double Maxx = 0;
 for ( int i = 0; i < n; i++ )
  if ( FreqTable[i][0] > Maxx)
      Maxx =    FreqTable[i][0];
 int Maxxx = (int) (deltaX*Maxx + deltaX);
 //The maximum y-coordinate is found
 int Maxyy = 0;
 for ( int i = 0; i < n; i++ )
  if (FreqTable[i][1] > Maxyy)
       Maxyy = (int) FreqTable[i][1] + 2;
 //Starting clause
 //x- and y-units.
 //The needed space is defined
 System.out.println("\\begin{center}" +
     "\n\\psset{xunit=0.15,yunit=0.3}" +
 "\n\\begin{pspicture}(" + Minxxx +",0)(" +
 Maxxx+ "," + Maxyy+")");
 //A rectangle is drawn for each entry in the
 //frequency table

 int lim1,lim2;
 for ( int i = 0; i < n; i++ )
  {
   lim1 = (int) (deltaX*FreqTable[i][0] -deltaX/2);
   lim2 = (int) (lim1 + deltaX);
   System.out.print(
 "\n\\psline("+ lim1 + ",0)" +
     "("+lim1 +"," + FreqTable[i][1]+ ")" +
     "("+lim2 + ","+ FreqTable[i][1]+ ")" +
     "("+lim2 + ","+ "0)"
    );
  }

 //Horizontal axis
 System.out.println("\n\\psline(" +
      Minxxx +",0)("+ Maxxx + ",0)");

 //Class markers (first coordinates) are indicated
 for ( int i = 0; i < n; i++ )
   {
int xposition = (int) (deltaX* FreqTable[i][0]);
int yposition = - 1;
    System.out.print
 (
  "\n\\rput*("+ xposition + "," + yposition+ "){"+
```

```
   (int) (FreqTable[i][0])+ "}"
 );
   }
 System.out.println();

 //Labels for heights (second coordinates) are indicated
 for ( int i = 0; i < n; i++ )
  {
   int xposition = (int) (deltaX* FreqTable[i][0]);
   int yposition = (int) (1+ FreqTable[i][1]);
   System.out.print
(
  "\n\\rput*("+ xposition + "," + yposition+ "){" +
      (int) (FreqTable[i][1])+ "}"
 );
 }

 //End clause
 System.out.println( "\n\\end{pspicture}");
 System.out.println( "\\end{center}");
 }

public static void codePSTricks()
 {
  System.out.println("\n\nPSTRICKS CODE FOR A BAR DIAGRAM");
  double  FreqT[][] =
        {{3, 3},
         {4, 3},
         {5, 2},
         {6, 2},
         {7, 1},
         {8, 1}};
 printMatrix(FreqT, "\n\nFREQUENCY  TABLE");
 makeCode(FreqT);
 separator();
 }



//=================================================
//        BINOMIAL DISTRIBUTION
//=================================================

//The  binomial distribution
//for  n repeats
```

```
//probability of head p,
//amplification factor F.
//Arithmetic trickery in action.
private static double[]  binomialDistribution(int n,
          double p, double F  )
{
 double pBinomial[] = new double[n+1];
 double pBinomialF[] = new double[n+1];
 double CoefficientsBinomial[] = new double[n+1];
 double biCoeff = 1;
 CoefficientsBinomial[0] = biCoeff;
 for(int r = 1; r < n+1; r++)
  {
   biCoeff = biCoeff * (n+1-r) / r;
   CoefficientsBinomial[r] = biCoeff;
  }
 System.out.println("\nCoefficients ");
 printVector(CoefficientsBinomial,n+1);
 System.out.println("Sum = " +
     sumVector(CoefficientsBinomial,n+1));

 for(int r = 0; r < n+1; r++)
  {
   pBinomial[r] = CoefficientsBinomial[r]  *
   Math.exp( r* Math.log(p) + (n-r) * Math.log(1-p));
  }
 System.out.println("\nProbabilities for 0,1,2...");
 System.out.println("Paste this vector to Excel" +
           " and make a graphic");
 printVector(pBinomial,n+1);
 sumVector(pBinomial,n+1);
 System.out.println("Sum = "
            +  sumVector(pBinomial,n+1));
 for(int k = 0; k < n+1; k++)
 pBinomialF[k] =  F* pBinomial[k];
 System.out.println("\nAbsolute frequencies for F = " + F);
 printVector(pBinomialF,n+1);
 System.out.println("Sum = "  +
           sumVector(pBinomialF,n+1));
 return pBinomial;
}

//Probability of closed region (extremes included)
private static double pClosedRegion( int limitInf,
                              int limitMax,
  double[] pBinomial)
```

```
{
 double  sum =0;
 for(int r = limitInf; r <= limitMax; r++)
  sum =  sum +  pBinomial[r];
 return sum;
}

//Detailed computation of the mean
private static void detailedMean(int n,
  double p,
  double[] pBinomial)
{
 System.out.println("\nDetailed mean: xp(x)");
 double  sum =0;
 double term;
 for(int r = 0; r < n+1; r++)
 {
  term =  r *  pBinomial[r];
  System.out.println( r + "    " + term) ;
  sum = sum +term;
 }
 System.out.println( "mean = sum xp(x) = " + sum) ;
 System.out.println( "Expected mean = np =  " + n*p) ;
}

//Detailed computation of the variance
private static void detailedVariance(int n,
  double p,
  double[] pBinomial)
{
 System.out.println("\nDetailed variance: " +
  "(x-mean)*(-mean)*p(x)");
 double  sum =0;
 double term;
 for(int r = 0; r < n+1; r++)
 {
  term =  Math.pow(r- n*p,2) *  pBinomial[r];
  System.out.println( r + "    " +  term) ;
  sum = sum +term;
 }
 System.out.println( "Variance = " +
"sum (x-mean)*(-mean)*p(x) = " + sum) ;
 System.out.println(
"Expected var = np(1-p) =  " + n*p*(1-p)) ;
 System.out.println( "Deviation = " + Math.sqrt(sum));
}
```

```
//Critical value for significance level alpha, one tail
private static void  critical1(  double[] pBinomial,
                                 double alpha )
{
 int numbTails = 1;
 System.out.println("\nLower tail");
 double cumulative = 0;
 double target = alpha;
 int i=-1;
 while( cumulative < target)
 {
  i = i+1;
  cumulative = cumulative + pBinomial[i];
  System.out.println(i + "  cumulative = " + cumulative);
 }
 System.out.println("To cumulate in the lower tail "
    + target
+ ", \nwe need the first " + i  + " Terms.");
 System.out.println("The critical value for alpha =  "
   + alpha
   + ", numb of tails = " + numbTails
   + "\nin the lower tail is "
   + i+ ". All terms below it are extreme.\n");
 System.out.println("Upper tail");
 int n = pBinomial.length;
 i= n;
 cumulative = 0;
 while( cumulative < target)
  {
   i = i-1;
   cumulative = cumulative + pBinomial[i];
   System.out.println(i + "  cumulative = " + cumulative);
  }

 System.out.println("To cumulate in the upper tail "
    + target
+ ", \nwe need term  " + i  + " and above.");
 System.out.println("The critical value for alpha =  "
   + alpha
   + ", numb of tails = " + numbTails
   + "\nin the upper tail is "
   + i + ". All terms above it are extreme.\n");
}
```

```
//Critical value for significance level alpha, two tails
private static  void  critical2(double[] pBinomial,
                            double alpha)
{
 critical1(pBinomial,alpha/2);
}


//Main of binomial distribution
private static void binomialDist(int n, double p, int F)
{
 double pBinomial[] = new double[n+1];
 System.out.println("n= " + n);
 System.out.println("p = " + p);
 System.out.println("F = " + F);
 pBinomial = binomialDistribution(n,p, F);
 detailedMean(n,p,pBinomial);
 detailedVariance(n,p,pBinomial);
 //Probability of a closed region (extremes included)
 int limitInf = 2;
 int limitMax = 4;
 double pR = pClosedRegion(limitInf,limitMax,pBinomial);
 System.out.println( "\nProb of closed region in " +
"within "+ limitInf + "  and " + limitMax +
 ",\nextremes included = " + pR) ;

}

//Critical values for a given
//significance level
private static void criticalValues(int n, double p,
                            double alpha)
{
 double pBinomial[] = new double[n+1];
 pBinomial = binomialDistribution(n,p, 1);
 System.out.println("Binomial distribution: \n" +
" n = " + n + ", p = " + p + ", " +
"significance level = alpha = " + alpha);
 System.out.println("\nCRITICAL VALUES");
 System.out.println("\nONE TAIL, alpha = " +alpha);
 System.out.println("\nTerms and cumulative values");
 critical1(pBinomial,alpha);
 //critical1(n,p, alpha);
 System.out.println("\nTWO TAILS, alpha = " +alpha);
 System.out.println("\nTerms and cumulative values");
```

```
 critical2(pBinomial, alpha);
 //critical2(n,p, alpha);
}

//Binomial distribution
public static void binomialDistribution()
{
 System.out.println("\n\nBINOMIAL DISTRIBUTION ");
 //Example 44 pag 32.
 int n= 6;
 double p = 0.25;
 int F = 35;
 //binomialDist(n,p,F);
 //Inverse problem: find critical values
 double alpha = 0.02;
 criticalValues(n,p,alpha);
 separator();
}



//=================================================
// THE TOP 12  TESTS
//=================================================


//============Machinery of general use

//This is constant pi
private static final double  pi = 3.14159265358979323846;

//Symbolic initialization that can be modified
//Significasnce level
private static double alpha = 0.05;
//Degrees of freedom of the t distribution
private static double dfT = 9;
//Degrees of freedom of the chi2 distribution
private static double dfChi2 = 17;
//Degrees of freedom of the F distribution, numerator
private static double dfFNum = 4;
//Degrees of freedom of the F distribution, denominator
private static double dfFDen = 32;

//====== Methods of general use=========
```

```
//Returns numb!
private static long factorial (long numb)
{
 if (numb <=1 ) return 1;
else
 return numb * factorial (numb -1);
}

//Returns the double factorial
private static double doubleFact(double x)
{
 if (x <=0 ) return 1;
else
 return x * doubleFact(x -2);
}


//Returns the Gamma function
private static double Gamma(double x)
{
 int xRounded = (int) x;
 //System.out.println( x + "   " + xRounded);
 if (xRounded == x)
return factorial (xRounded-1);
 else
 {
 return Math.sqrt(pi)* doubleFact(2*xRounded -1) /
          Math.pow(2,xRounded);
 }
}



//======Inner top class that calculates==========
//============critical and p-values ==============

//The methods that eventually can be useful
//to all distributions come here such as they are.
//The methods that are privative of each distribution
//are defined here as abstract
//and implemented in  specific classes below.
abstract class ValueAbstract
{
double sumzero;
double Error;
double jCritic;
```

```
double k;



//The density function is of abstract type.
//It is not implemented,
//so we can implement other functions too and
//process them with this very same machinery.
abstract double f(double t);



//========= p-values==========

//Returns the integral under the function f
//in within 0 and z. Precision=1/N.
//Simpson's rule adapted for reuse.
double HalfBodySimpson(double z, long N)
{
 double h = z/(2*N);
 long m = N;
 double sum = f(0);
 for(int j= 1; j<=m; j++)
 sum = sum +   4* f((2*j-1) * h);
 for(int j= 1; j<m; j++)
 sum = sum +   2* f((2*j) * h);
 sum = sum + f((2*m) * h);
 sum = (h/3)*sum;
 return sum;
}


//The area in within zero and z of chosen
//distribution is calculated
//for various degrees of precision.
double area(double z)
{
 z = (Math.sqrt(1+4*z*z)-1)/(2*z);
 int N;
 double  answer;
 int i = 3;
 N = (int) Math.pow(10, i);
 answer = HalfBodySimpson(z,N);
 return answer;
}
```

```
//Returns Psi(e) = p( x < e)
//the cumulative distribution function of e.
//This is abstract because it depends
//on the distribution, it must be
//implemented in the corresponding classes.
double psi(double e)
{
 return 0.5 + area(e);
}

//Returns the p-value(e) = p( x > e)
//Local implementation of the p-value of  e
double pValue(double e)
{
 return 1- psi(e);
}


//Local manager
void pExecutor(double testValue)
{
 System.out.println("\nTest value = " + testValue);
 System.out.print("Simpson's rule."+
  "\nArea = p( 0 <  x < " + testValue + ") = ");
 double area =  area(testValue);
 System.out.println( area);
 //psi(e) = p( x < e)
 double psi = psi(testValue);
 System.out.println("psi(" + testValue + ") " +
  "= p( x < " + testValue + ") = " +  psi );
 double pValue = pValue(testValue);
 System.out.println("p-value(" +
      testValue + ") = " +
    "p( x > " + testValue + ") = " + pValue);
}

//========Critical values=====

//Reports the transformed upper u-critical for
//the given distribution.
//Significance level = alpha.
//Precision=1/N.
//Simpson's rule adapted for reuse.
//The error of approximation is also calculated.
double upperU(double h, double target)
{
```

```
 double r = h/6;
 jCritic = 0;
 double sum = sumzero;
 for(int j= 0; sum < target; j++)
  {
sum = sum +
    r*(f( j*h) + 4* f((j+0.5) * h) +  f((j+1) * h));
jCritic = j+1;
   }
 jCritic = jCritic-1;
 double greatSum = sum -
    r*(f( jCritic*h) + 4* f((jCritic+0.5) * h)
         +  f((jCritic+1) * h));
 Error = target-greatSum;
 double uAprox = (jCritic) *h;
 return uAprox;
}



//Returns the transformed value z = u/(1-u*u)
double z(double u)
{
 double z = u/(1-u*u);
 return z;
}

//The wide of subintervals is defined
double h()
{
 double u = 1;
 int i=3;
 long N = (int) Math.pow(10, i);
 double h = u/N;
 return h;
}

//Reports the upper z-critical for
//Significance level = alpha.
double vCrit(double target)
{
 double h = h();
 //System.out.println( "h = " + h);
 double uCritic =  upperU(h, target);
 double zCritic = z(uCritic);
 return zCritic;
```

```
}


//*****Analytical hunting*******

//Returns the density function of the distribution
abstract double ff(double z);



//Returns F(a,b) = h(f(a)  + 4 * f((a+b)/2)  + f(b))-Error;
double F(double a, double b)
{
 double h = (b-a)/6;
 double  Fb = h* (  ff(a) + 4 * ff((a+b)/2)
                  + ff(b)  )-Error;
 return Fb;
}


//Returns the  ratio  F(zCritic,gi)/ ff(gi)
double ratio(double zCritic, double gi)
{
 double  ratio =  F(zCritic,gi)/ ff(gi) ;
 return ratio;
}

//Critical value for the upper tail.
//A professional  approximation is found by
//analytical hunting
double hunting(double zCritic )
{
 double h = h();
 double ur =  (jCritic+1)*h;
 //Right border of the critical interval
 double rBorder = z(ur);
 double gSubi = rBorder;
 double gSubiPlus= 0;
 int iterations = 0;
 int maxIterations = 30;
 boolean more = true;
 while (more)
  {
gSubiPlus = gSubi -  ratio(zCritic, gSubi);
iterations = iterations +1;
if (iterations >= maxIterations)  more = false;
```

```
if (gSubiPlus == gSubi) more = false;
gSubi = gSubiPlus;
  }
 return gSubiPlus;
}

//Critical value for significance level alpha, one tail
double  critical(  double alpha)
{
 double target = 1-alpha;
 double vCritic = vCrit(target);
 vCritic = hunting(vCritic);
 return vCritic;
}

//Local manager
void critManager(double alpha)
{
System.out.println( "\nAlpha = Significance level = "
+ alpha + ".");

System.out.println("Critical values for ONE TAIL = +/-"
+  critical(alpha) );

double alphaHalf = alpha/2;
System.out.println("Critical values for TWO TAILS = +/-"
            + critical(alphaHalf) );
}

}//end of class ValueAbstract


//======Particular classes========


//The z class.
//Definition of the z density function
//and exclusive to it methods.
//It is an specification of the ValueAbstract class.
class VzClass extends ValueAbstract
{

 //Constructor: this is the first thing to do
 VzClass()
 {
  sumzero = 0.5; //half bell
```

```
  System.out.println("\n===== z distribution");
}


//Function f is implemented here.
//Returns the transformed density function of the
//standardized normal distribution.
double f(double t)
{
 double z = t/(1-t*t);
 double  f = 1 /Math.pow(2*pi, 0.5)*Math.exp(-z*z/2)*
             (1+ t*t)/( (1-t*t)*(1-t*t));
 return f;
}


//Returns the density function of the Z distribution
double ff(double z)
{
 double  f =  Math.exp(-z*z/2)/Math.pow(2*pi, 0.5);
 return f;
}

}//end of class VzClass


//The t class
//Definition of the t density function
//and exclusive to it methods.
//It is an specification of the ValueAbstract class.
class VtClass extends ValueAbstract
{

//Constructor: this is the first thing to do
VtClass()
{
 sumzero = 0.5; //half bell
 System.out.println("\n===== t distribution");
 System.out.println("t degrees of freedom = " + dfT);
}



//Function f is implemented here.
//Returns the transformed density function of the
//t distribution with dfT d.f.
```

```
double f(double t)
{
 double nu = dfT;
 double z = t/(1-t*t);
 double  f = ( Gamma((nu+1)/2)
   / ( Math.sqrt(pi*nu)*Gamma(nu/2)) )
   * Math.pow(1+z*z/nu, -(nu+1)/2 )
                  * (1+ t*t)/( (1-t*t)*(1-t*t));
 return f;
}


//Returns the density function of the t-distribution
double ff(double z)
{
 double nu = dfT;
 double  f =  ( Gamma((nu+1)/2)
    /  ( Math.sqrt(pi*nu)*Gamma(nu/2)) )
    *  Math.pow(1+z*z/nu, -(nu+1)/2 ) ;
 return f;
}

}//end of class VtClass



//The chi2 class
//Definition of the Chi2 density function
//and exclusive to it methods.
//It is an specification of the ValueAbstract class.
class VChi2Class extends ValueAbstract
{

//Constructor: this is the first thing to do
VChi2Class()
{
 sumzero = 0; //no negative values
 System.out.println("\n===== Chi2 distribution");
 System.out.println("Chi2 degrees of freedom = " + dfChi2);
}

//Returns Psi(e) = p( x < e)
//the cumulative distribution function of e
double psi(double e)
{
 return area(e);
```

```
}

//Function f is implemented here.
//Returns the transformed density function of the
//chi2 distribution with nu d.f.
double f(double t)
{
 double nu = dfChi2;
 double z = t/(1-t*t);
 double  f = ( 1 /
   ( Math.pow(2, nu/2)*Gamma(nu/2)) )
   * Math.exp(-z/2)* Math.pow(z, (nu-2)/2)
   * (1+ t*t)/( (1-t*t)*(1-t*t));
 return f;
}



//Returns the density function of the chi2-distribution
double ff(double z)
{
 double nu = dfChi2;
 double  f = (1/(Math.pow(2, nu/2)*Gamma(nu/2)))
             * Math.exp(-z/2)* Math.pow(z, (nu-2)/2);
 return f;
}


//Local manager
void manager(double alpha)
{
 System.out.println("Critical values for ONE TAIL "
    + "\nof the chi-square- "
+  "distribution with df =  "
+ dfChi2
+ "\nAlpha = Significance level = "
+ alpha + ".");

 System.out.println("Lower critical value  = "
+  critical(1-alpha) );
 System.out.println("Upper critical value  = "
+  critical(alpha) );
 System.out.println("\nCritical values for TWO TAILS "
    + "\nof the chi-square "
+  "distribution with df=  "
+ dfChi2
```

```
+ "\nAlpha = Significance level = "
+ alpha + ".");
 double alphaHalf = alpha/2;
 double Chi2Menos = critical(1-alphaHalf) ;
 System.out.println("Lower critical value  = "
+  Chi2Menos  );
 double Chi2Mas = critical(alphaHalf) ;
 System.out.println("Upper critical value  = "
+  Chi2Mas  );
}
}//end of class pVChi2Class



//The F class
//Definition of the Chi2 density function
//and exclusive to it methods.
//It is an specification of the ValueAbstract class.
class VFClass extends ValueAbstract
{

//Constructor: this is the first thing to do
VFClass()
{
 sumzero = 0; //no negative values
 double nu1 = dfFNum;
 double nu2 = dfFDen;
 double a =  Math.pow(nu1, nu1/2);
 double b =  Math.pow(nu2, nu2/2);
 double c = beta(nu1/2,nu2/2);
 k = a* b / c;
 System.out.println("\n===== F distribution");
 System.out.println("F degrees of freedom: "
           + "Num = " + dfFNum
           + ", Den = " + dfFDen);
}

//Returns Psi(e) = p( x < e)
//the cumulative distribution function of e
double psi(double e)
{
 return area(e);
}
```

```
//Returns the transformed density function of the
//F distribution with dfFNum and dfFDen d.f.
double f(double t)
{
 double nu1 = dfFNum;
 double nu2 = dfFDen;
 double z = t/(1-t*t);
 double  f =   k*Math.pow(z, nu1/2-1)
            * Math.pow(nu2+nu1*z, -(nu1+nu2)/2)
            * (  (1+ t*t)/( (1-t*t)*(1-t*t))  ) ;
 //System.out.println(f);
 return f;
}


//Returns a helping  function  with   d.f. =
//a(num) and b(den)
double g(double x, double a, double b)
{
 double  g = Math.pow(x, a -1)*
                 (Math.pow(1-x, b -1));
 return g;
}


//Returns the integral under the function f
//in within 0 and u. Precision=1/N.
//Simpson's rule adapted for reuse.
//a and b are degrees of freedom
double HalfBodySimpsong(double u,
double a, double b, long N)
{
 double h = u/(2*N);
 long m = N;
 double sum = g(0,a,b);
 for(int j= 1; j<=m; j++)
 sum = sum +   4* g((2*j-1) * h,a,b);
 for(int j= 1; j<m; j++)
 sum = sum +   2* g((2*j) * h,a,b);
 sum = sum + g((2*m) * h,a,b);
 sum = (h/3)*sum;
 return sum;
}


//Returns the  beta  function in integral form
```

```
//with a and b d.f.
double beta( double a, double b)
{
 double  answer;
 int i = 4;
 int  N = (int) Math.pow(10, i);
 double u=1;
 answer = HalfBodySimpsong(u,a,b,N);
 return answer;
}


//Reports the transformed upper u-critical for
//the given distribution.
//Significance level = alpha.
//Precision=1/N.
//Simpson's rule adapted for reuse.
//The error of approximation is also calculated.
double upperU(double h, double target)
{
 double r = h/6;
 double sum = 0;
 jCritic = 0;
 for(int j= 0; sum < target; j++)
  {
   sum = sum +
    r*(f( j*h) + 4* f((j+0.5) * h) +  f((j+1) * h));
jCritic = j+1;
  }
 jCritic = jCritic-1;
 double greatSum = sum -
           r*(f( jCritic*h) + 4* f((jCritic+0.5) * h)
         +  f((jCritic+1) * h));
 Error = target-greatSum;
 double uAprox = (jCritic) *h;
 return uAprox;
}


//Returns the density function of the t-distribution
double ff(double z)
{
 double nu1 = dfFNum;
 double nu2 = dfFDen;
 double  f = k*Math.pow(z, nu1/2-1)
  * Math.pow(nu2+nu1*z, -(nu1+nu2)/2)   ;
```

```
 return f;
}

//Local manager
void manager(double alpha )
{
 System.out.println("\nF distribution");
System.out.println("Critical values for ONE TAIL "
    + "\nof the F- "
+ "distribution with df:  "
+ dfFNum + " and " + dfFDen+ ", "
+ "\nAlpha = Significance level = "
+ alpha + ".");

System.out.println("Lower critical value  = "
+  critical(1-alpha) );
System.out.println("Upper critical value  = "
+  critical(alpha) );

System.out.println("\nCritical values for TWO TAILS "
    + "\nof the f  "
+ "distribution."
+ "\nAlpha = Significance level = "
+ alpha + ".");
double alphaHalf = alpha/2;

System.out.println("Lower critical value  = "
+  critical(1-alphaHalf) );
System.out.println("Upper critical value  = "
+  critical(alphaHalf) );
}
}//end of class pVFClass


//=======================================
//======== THE 12 MOST USUAL TESTS ======
//=======================================


//======= 1: z-test for one mean ========

//Runs a z-test for the mean
//knowing the mean of a random sample
//and the populational variance.
//Confidence interval for the mean is also calculated.
```

```
static void printData1(double xBar, double mu,
double sigma, double n,  double alpha)
{
System.out.println("\nz Test to study the null" +
 "\nhypothesis about a mean  given the mean " +
 "\nof a random sample and the standard deviation" +
 "\nof the population.\n");
System.out.println("Observed mean = xBar = " + xBar);
System.out.println("Ho: expected mean = mu = " + mu);
System.out.println("sigma = " + sigma);
System.out.println("n = " + n);
System.out.println("alpha = " + alpha);
}


/*Measures the discrepancy between observed and expected
 * in  z Test to study the null hypothesis about a mean
* given the mean of a sample and the standard deviation
* of the population.*/
static double  zExpOneMean(double xBar, double mu,
double sigma, double n)
{
 System.out.println(
  "zExp = (xBar- mu)/( sigma/Math.sqrt(n))");
 double zExp = (xBar- mu)/( sigma/Math.sqrt(n));
 return zExp;
}

/*Confidence interval for the mean
given the populational variance */
static void confIntMeanZ(double xBar, double mu,
double sigma, double n,  double alpha)
{
System.out.println( "\nConfidence interval of the mean" +
" \ngiven the populational deviation sigma");
double alphaHalf = alpha/2;
double zc2 =  p.critical(alphaHalf);
double r = zc2*sigma / Math.sqrt(n);
System.out.println( "r = zc2*sigma / Math.sqrt(n)" );
System.out.printf( "r = %10.4f \n", r );
double limInf = xBar - r;
double limSup = xBar + r;
```

```
System.out.printf( "limInf = xBar - r " +
                      "= %10.4f \n" , limInf );
System.out.printf( "limSup  = xBar + r " +
                      "= %10.4f \n", limSup );
}

static //Standard error of the mean
double sError(double sigma, double n)
{
System.out.println("\nStandard error  = " +
"sigma/Math.sqrt(n)");
return sigma/Math.sqrt(n);
}

static //Needed n = size of the sample
//given tolerance E in the evaluation of the mean.
int nNeeded(double sigma, double E)
{
System.out.println("\nTolerance error  E = "+ E);
//Critical z with two tails
double alphaHalf = alpha/2;
double zc2 =  p.critical(alphaHalf);
System.out.println("m= " + "zc2 * sigma / E ");
//Needed n
double m = zc2 * sigma / E;
System.out.println("Needed  n = m*m");
//Upwards approximation
return (int) Math.ceil(m*m);
}



//Runs a z test for one mean
public static void zTestOneMeanWork(double xBar, double mu,
double sigma, double n,  double alpha )
{

//Top class is instructed to take
//the form of the particular class
p = z;
//z-test for one mean
printData1(xBar,mu,sigma,n,alpha);
//Critical values
double zCriticOneTail = p.critical(alpha);
double alphaHalf = alpha/2;
```

```
double zCriticTwoTails = p.critical(alphaHalf);
//z-test
System.out.printf( "Discrepancy between observed and ");
System.out.printf( "expected: \n");
double zExp = zExpOneMean(xBar,mu,sigma,n);
System.out.printf( "zExp = %10.4f \n" , zExp);
double pValue = p.pValue(zExp);
System.out.println("p-value(" +
      zExp + ") = " +
    "p( x > " + zExp + ") = " + pValue);
System.out.printf("zCritic for TWO TAILS " +
"= zc2 =  +/-%10.4f \n",  zCriticTwoTails);
System.out.printf("zCritic for ONE TAIL  " +
"= zc1 = +/-%10.4f \n",  zCriticOneTail);
//Standard error
double error = sError(sigma,n);
System.out.printf("Standard error  = %10.4f \n", error);
//Confidence interval
confIntMeanZ(xBar,mu,sigma,n,alpha);
//Admitted tolerance for mu
double E = 5;
//Size of sample to attain tolerance E
int nNeeded = nNeeded(sigma,  E);
System.out.printf("Needed n = %10d \n", nNeeded);
separator();
}

//Data input for  a z test for one mean
public static void zTestOneMean( )
{
//Example 184, pg 169
double xBar = 40;
double mu = 20;
double sigma = 16;
double n = 36;
alpha = 0.05;
zTestOneMeanWork(xBar,mu,sigma,n,alpha);
}

}//End of main class
```

107. ***Exercise.*** *Run the program and play with the code.*

5.2 Popular experiments

We have included in our toolBox the statistics for a first experiment. To include others is just more of the same. So, let us list the most usual formats for experiments.

108. ***The top 12***

The next is a list of the 12 most usual formats for experiments. We follow our text associated to this series on Elementary Statistics Vol I version 7.

1. **Study of one mean knowing σ.** Done.

2. **Comparison of the means of X and Y knowing σ_X, σ_y,**

   Two tails test: $H_o : \mu_x - \mu_Y = 20$ and $H_a : \mu_x - \mu_Y \neq 20$

   One tail test: $H_o : \mu_x - \mu_Y = 20$ and $H_a : \mu_x - \mu_Y < 20$

   Stadigraph to measure the discrepancy between observed difference $\bar{X_m} - \bar{Y_n}$ and expected one $\mu_X - \mu_Y$ according to H_o.

   $$z = \frac{(\bar{X_m} - \bar{Y_n}) - (\mu_X - \mu_Y)}{\sqrt{\frac{{\sigma_X}^2}{m} + \frac{{\sigma_Y}^2}{n}}}$$

   Confidence interval for $\mu_X - \mu_Y$:

   $$(\bar{X_m} - \bar{Y_n}) - z_{(\alpha,2)}\sqrt{\frac{{\sigma_X}^2}{m} + \frac{{\sigma_Y}^2}{n}} < \mu_X - \mu_Y < (\bar{X_m} - \bar{Y_n}) + z_{(\alpha,2)}\sqrt{\frac{{\sigma_X}^2}{m} + \frac{{\sigma_Y}^2}{n}}$$

3. **Estimation of a mean knowing the deviation of a sample.**

   $\widehat{\sigma_X^2} = s = \sqrt{\frac{\Sigma(x_i - \bar{X})^2}{n-1}} = \sqrt{\frac{n(\Sigma x^2) - (\Sigma x)^2}{n(n-1)}}$

   Two tails test: $H_o : \mu_x = 20$ and $H_a : \mu_x \neq 20$

   One tail test: $H_o : \mu_x = 20$ and $H_a : \mu_x < 20$

   Stadigraph to measure the discrepancy between observed $\bar{X_n}$ and the expected μ_X according H_o :

   $t = \frac{\bar{X_n} - \mu_X}{\frac{\widehat{\sigma_X}}{\sqrt{n}}}$

4. **Study of one proportion**.

   - La $H_o : p = p_o$

     Stadigraph to measure the discrepancy between observed relative frequency f and expected proportion p_o according to H_o:

     $$z = \frac{f - p_o}{\sqrt{\frac{p_o(1-p_o)}{n}}}$$

- Confidence interval of $1-\alpha$ for p. If $f=\frac{k}{n}$, the interval is
$$f-(z_{(\alpha,2)})\sqrt{\frac{f(1-f)}{n}}<p<f+(z_{(\alpha,2)})\sqrt{\frac{f(1-f)}{n}}$$
- Needed number of registered events to asses p_o when tolerance of E is admitted:
$$n=(z_{(\alpha,2)})^2\frac{1}{4E^2}.$$

5. **Comparison of two proportions that come from independent samples:**

Two tails test: $H_o: p-r=-0.20$ and $H_a: p-r\neq -0.2020$

One tail test: $H_o: p-r=-0.20$ and $H_a: p-r<-0.20$

Stadigraph to measure the discrepancy between observed difference $f-g$ and expected one $p-r$ according to H_o:

$$z=\frac{(f-g)-(p-r)}{s_b}$$

where the binomial deviation s_b can take one among various forms:

Case 1: If p and r are given together with the sizes of samples m, n

$$s_b=\sqrt{\frac{p(1-p)}{m}+\frac{r(1-r)}{n}}$$

Case 2: When the expected difference is zero, $p-r=0$, we can join all data into a single sample to estimate p:

$$\hat{p}=\frac{X+Y}{m+n}$$

One can also use the alternate formula:

$$\hat{p}=\frac{fm+gn}{m+n}$$

$$s_b=\sqrt{\hat{p}(1-\hat{p})(\frac{1}{m}+\frac{1}{n})}$$

Case 3: When p and r are not given but the expected difference $p-r$ is known together with relative frequencies f, g, and the sizes of samples m, n:

$$s_b=\sqrt{\frac{f(1-f)}{m}+\frac{g(1-g)}{n}}$$

Confidence interval for the difference of proportions $p-r$:

$$f-g-(z_{(\alpha,2)})\sqrt{\frac{f(1-f)}{m}+\frac{g(1-g)}{n}} < p-r < f-g+(z_{(\alpha,2)})\sqrt{\frac{f(1-f)}{m}+\frac{g(1-g)}{n}}$$

6. **Study of one variance**

   Two tails test: $H_o : \sigma^2 = 200$ and $H_a : \sigma^2 \neq 200$

   One tail test: $H_o : \sigma^2 = 200$ and $H_a : \sigma^2 < 200$

   Stadigraph to measure the discrepancy between observed $\widehat{\sigma^2}$ and expected σ^2 according to H_o:

$$\chi^2 = \frac{(n-1)\widehat{\sigma^2}}{\sigma^2} \text{ con } n-1 \text{ g.l.}$$

   Confidence interval for the variance σ^2

$$\frac{(n-1)\widehat{\sigma^2}}{\chi^2_+} < \sigma^2 < \frac{(n-1)\widehat{\sigma^2}}{\chi^2_-}$$

7. **Comparison of two variances.**

   We use a ratio between large and small variances.

   $H_o : \frac{\sigma^2_L}{\sigma^2_S} = R$

   $R_{exp} = \frac{\widehat{\sigma^2_L}}{\widehat{\sigma^2_S}}$

   with $m-1$ d.f in the numerator and $n-1$ in the denominator.

   Stadigraph to measure the discrepancy between observed R_{exp} and expected R:

   $F = \frac{R_{exp}}{R} = \frac{\widehat{\sigma^2_G}}{R\widehat{\sigma^2_P}}$

   with $m-1$ d.f in the numerator and $n-1$ in the denominator.

   Confidence interval for the ratio between two variances $\frac{\sigma^2_G}{\sigma^2_P}$:

$$\frac{\widehat{\sigma^2_G}}{\widehat{\sigma^2_P}F_+} < \frac{\sigma^2_G}{\sigma^2_P} < \frac{\widehat{\sigma^2_G}}{\widehat{\sigma^2_P}F_-}$$

   where $F_+ = F_{+,\mu,\nu}$ with df of numerator $= \mu$ and df of denominator $= \nu$.

   $F_- = F_{-,\mu,\nu} = \frac{1}{F_{+,\nu,\mu}}$

8. **Comparison of two means knowing after the study of variances that populational variances are equal.**

   Stadigraph to measure the discrepancy between observed $\bar{X_m} - \bar{Y_n}$, and expected $\mu_X - \mu_Y$ according to H_o:

$$t = \frac{(\bar{X_m}-\bar{Y_n})-(\mu_X-\mu_Y)}{\widehat{\sigma_J}\sqrt{\frac{1}{m}+\frac{1}{n}}}$$

Join deviation results from uniting the two samples into a single one to estimate the only one deviation $\widehat{\sigma_J}$:

$$\widehat{\sigma_J} = \sqrt{\frac{\widehat{\sigma_X}^2(m-1)+\widehat{\sigma_Y}^2(n-1)}{n+m-2}}$$

Confidence interval for $\mu_X - \mu_Y$:

$$(\bar{X_m} - \bar{Y_n}) - t_{(\alpha,2)}\widehat{\sigma_J}\sqrt{\frac{1}{m}+\frac{1}{n}} < \mu_X - \mu_Y < (\bar{X_m} - \bar{Y_n}) + t_{(\alpha,2)}\widehat{\sigma_J}\sqrt{\frac{1}{m}+\frac{1}{n}}$$

df= n+m-2.

9. **Comparison of two means knowing by the previous study of variances that populational variances are different.**

   Observed $\bar{X_m} - \bar{Y_n}$

   Expected $H_o : \mu_X - \mu_Y = 30$

   Stadigraph to measure the discrepancy between observed, $\bar{X_m} - \bar{Y_n}$, and expected $\mu_X - \mu_Y$ according to H_o:

   $$t = \frac{(\bar{X_m} - \bar{Y_n}) - (\mu_X - \mu_Y)}{\sqrt{\frac{\widehat{\sigma_X}^2}{m} + \frac{\widehat{\sigma_Y}^2}{n}}}$$

   Confidence interval for $\mu_X - \mu_Y$

   $$(\bar{X_m} - \bar{Y_n}) - t_{(\alpha,2)}\sqrt{\frac{\widehat{\sigma_X}^2}{m} + \frac{\widehat{\sigma_Y}^2}{n}} < \mu_X - \mu_Y < (\bar{X_m} - \bar{Y_n}) + t_{(\alpha,2)}\sqrt{\frac{\widehat{\sigma_X}^2}{m} + \frac{\widehat{\sigma_Y}^2}{n}}$$

   Degrees of freedom:

   $$g.l = \frac{(\frac{Var_X}{n_X} + \frac{Var_Y}{n_Y})^2}{\frac{(\frac{Var_X}{n_X})^2}{n_X+1} + \frac{(\frac{Var_Y}{n_Y})^2}{n_Y+1}} - 2$$

10. **Comparison of means for paired data:**

    $H_o : \mu_X - \mu_Y = 8$

    Data come in n pairs (x_i, y_i). Each pair gives raise to a signed difference $D_i = x_i - y_i$. The D_i have a mean $\bar{D}$ and a deviation s_D:

    $\bar{D} = \frac{x_i - y_i}{n}$.

    Stadigraph to measure the discrepancy between observed $\bar{D}$ and expected $\mu_X - \mu_Y$ according to H_o, :

    $$t = \frac{\bar{D} - (\mu_X - \mu_Y)}{\frac{s_D}{\sqrt{n}}}$$

The degrees of freedom of the t are $n-1$.

Confidence interval for difference of means for paired data:

$$\bar{D} - t_{(\alpha,2)}\frac{s_D}{\sqrt{n}} < \mu_X - \mu_Y < \bar{D} + t_{(\alpha,2)}\frac{s_D}{\sqrt{n}}$$

11. **Independence of attributes and contingency tables with data $Observed_{ij}$ in row i and column j.**

    Expected frequency under independence of attributes:

    $Expected_{ij} = \frac{(RowSum_i)\times(ColumnSum_j)}{GrandSum}$

    $Discrepancy = \sum \frac{(Observed_{ij}-Expected_{ij})^2}{Expected_{ij}} = (\sum \frac{(Observed_{ij})^2}{Expected_{ij}}) - n$

    The discrepancy must be compared with the critical χ^2 with $(r-1)(c-1)$ d.f.

12. **Multiple comparison of proportions:**

    Accommodate data as in a contingency table and apply the very same algorithm.

    H_o : proportions are homogeneous.

    H_a : there is at least one pair of different proportions.

109. ***Exercise.*** *Encode the top 12 experiments to graduate your toolBox7 to StatTrek version 1.*

5.3 Conclusion

We have merged all our particular tools into a great toolbox ready to be used. While we have enjoyed our work, it was full of challenges. In spite of the fact that our problem was of low to middle complexity, we have plainly verified that there is no software without a fierce battle with bugs.

Chapter 6

Conclusions

We have registered some few moments of the development of a project that was not precisely a homework for the next day. Our experience allows us to emphasize:

1. Our study has a purpose: it is specifically directed to show that the insight that **the genome is software and evolution is a software developer is both necessary and sufficient to deal with all fundamental questions about biological evolution**. Accepting that the software of the genome is as natural as the rest of the universe, then the study of evolution is a part of the study of software development in general. That is why our work in developing the software for StatTrek has been taken as a lab in evolution that we eagerly apply to biology. Some conclusions follow.

2. There are too many styles for programming most possibly one for each developer. But given that designing software is very difficult, it is our duty to learn a style that possibly would alleviate that difficult task. Our election has been to single out a gossipless, transparent, functional, OOP style of design. A **gossipless style** is one according to which local problems are solved locally using local variables, so design is not overburden with complexity. A **transparent style** is one in which the code, its output, the problem and its solution are all made on the same image. The importance of transparency stems form the fact that developing software is just going over fixing bugs whose correction generates more bugs. So, transparency easies the correction of bugs. A **functional style** is one that encapsulates a piece of code in a method if and only if it has a function with a clearly defined output in response to each input. The **OOP** (object oriented programming) style is one that encapsulates in a class ready for reuse a set of variables and methods if they are functionally associated as a unity.

3. A successful style produces **evolvable software**, which enables one to pro-

duce great buildings at fast pace by resorting to more elementary building blocks. Nevertheless, **evolutionary inertia** (the blind use of building blocks at hand) is not as efficient as considering that big problems need big strategies and not a sum of tiny ones.

4. We have witnessed that it is very easy to produce not evolvable software. Rather, one produces code that solves the problem of the day but if one desires to use this code to solve a problem of the following week, one sees that the previous code needs to be modified so much that it might be easier to design the needed code ab initio. This happens because one cannot grasp the role of global variables in a gossip style, or when one cannot see the function fulfilled by a piece of code making it not transparent, or when a piece of code has functions not listed in the output so that it is not functional, or when **encapsulation** is not faithful, dividing an item that function as a unity and merging things that function better if they are separated.

5. While we feel that evolvable software is akin to a gossipless, transparent, functional, OOP style, we know that this style is not a panacea. Say, we as yet have not solved the **problem of granulation**, which appears when one divides a task in too many small problems producing many methods that implement extremely simple functions. The arising trouble is that combining many elements is more difficult than combining some few ones. That is why one looks for a compromise appropriate to the given task: building blocks must be elementary like tiles but not too much as grains of sand.

6. We perceive that complexity breaks evolution: ordinary complexity is plenty of terrible problems that make fun of every style, say, if one carefully devises each method following the most purist style, at the moment of assembling a complex program, the ensemble looks as an entangled conglomerate that, because of mutual calls, one cannot understand how it functions. The arising situation is well known in the trade because one easily ends up with a code that resembles a 'big ball of mud', which 'is haphazardly structured, sprawling, sloppy, duct-tape and bailing wire, spaghetti code jungle' (Foote and Yoder, 1995). Therefore, one cannot use extant code to facilitate solving more complex problems.

7. We propose that a style results from a strategy to solve problems so it is a direct result of having a rather limited mind. Therefore, we predict that evolution must have no style, i.e., software devised by evolution must be dominated by disorder but locally it must follow a given style that never will get repeated anyhow. We consider this prediction mandatory and we conclude that it is plainly falsified: genetic engineering is possible just because the genome follows all around an extremely transparent style.

8. Complexity implies in real terms that one cannot guarantee that produced code might be bugfree. Quite to the contrary, reuse usually allows the

unveiling of previously unseen bugs. That is why the record consigned in the present volume does not depicts a temporal chart of development but only a pedagogical one. In other words, to say that the present material is an example of evolution is unduly because the past has received many improvements and corrections dictated by the future.

9. We have merged all our particular tools into a great toolbox ready to be used but at own risk because of hidden bugs. While we have enjoyed our work, it was full of challenges. Thus and in spite of the fact that our problem was of low to middle complexity, we have plainly verified that there is no software without a fierce battle with bugs. Nevertheless, we have erased any trace of correction of bugs, just as the genome of all living and all extinct species seems to be.

10. We have illustrated two of our general claims: evolution might be a handicap for further construction of complexity and that evolution and good style do not go hand in hand. Therefore evolution of complex features must left a clear track towards perfection that must be apparent in the fossil record. Nothing like that is found in nature so this is a mandatory prediction of the evolutionary theory that is falsified.

11. Summing up, we content three main points. First: evolution is a great idea whose implementation by Java has included such wonderful tools as abstract classes and overriding. Second: that there is no one scientific reason under the sun to consider that life on Earth might be explained by the evolutionary theory of present scientific international literature. Third: the falsifications of that theory can be hardened and multiplied. So, everyone in our community has a hard work for a long time: to show beyond all criticisms that the evolutionary theory is scientifically witless else to make a point of it. Our choice is the first one.

Answers to exercises

Problems of Chapter 1

10, pag. 6. Complete instructions are finished with ";". Commentaries are initialized with // and can be ended with any sign. A line that declares the name of the class or of a method does not have ";", instead it has a key pair: {}.

13, pag. 7.

```
//Program F13 biography
//Same as Program   A11
//This is my first exercise,
//It prints my name and birthday.
//The name of this class or program is biography

public class biography
{
 public static void main(String[] args)
  {
   System.out.println("L.J., the Intelligent.");
   System.out.println("Birthday: March 5th.");
  }
}
```

Problems of Chapter 2

23 pag. 15. To make **tables**, use the method *printf* that means *print with format* (inherited from C):

```
int i = 5;
double a = i*i;
System.out.printf( "%7d  %11.5f  %n",
                       i,     a);
```

This piece of code declares two variables, the first int and the second double. Next, the printing instruction reads: print an int or discrete variable using 7 spaces and next a variable of type double or float (less precision that double) using 11 spaces, 5 of which are for the decimal part. The instruction ends by specifying which variables must be printed. Notice that with *println* we use the symbol + to concatenate strings while with *printf* we use commas to list variables. Also, the symbol for change of car is "\n" with *println* while it is "%n" with *printf*. If we need to print a string, one uses the modifier "%s".

24, pag. 15. Program to calculate mean and variance of a list of data using an alternate definition.

```
//Program F24 MeanVarList2
//This program computes the mean and
//variance of a list of data.
//Alternate method is used.

public class meanVarList2
{

//The variance of a list of data
private static void meanVarVector(double DataVect[])
 {
  int n = DataVect.length;
  double sumx = 0;
  double sumx2 = 0;
  System.out.printf("%n          x         x^2");
  for(int i = 0; i < n; i++ )
  {
   sumx = sumx + DataVect[i];
   sumx2 = sumx2 + DataVect[i]*DataVect[i];
   System.out.printf("%n %10.3f   %10.3f ",
   DataVect[i], DataVect[i]*DataVect[i] );
  }
  System.out.printf("%n Sum %6.3f   %10.3f ",
                sumx,        sumx2 );
  System.out.print("\n \n");
  System.out.println("n = number of data = " + n);
  System.out.println("sumx = " + sumx);
  double mean = sumx / n;
  System.out.println("Mean = sumx / n");
```

```
  System.out.println("Mean = " + mean);
  System.out.println("sumx2 = " + sumx2);
  System.out.println("variance = " +
   "(n * sumx2 - sumx*sumx)/ (n*( n-1))");
  double var = (n * sumx2 - sumx*sumx)/ (n*( n-1));
  System.out.println( "Variance = "+ var);
  double deviation = Math.pow( var, 0.5);
  System.out.println( "Deviation = "+ deviation);
  double coeffVar =  deviation/mean;
  System.out.println( "Coefficient of variation = "
                                + coeffVar);
  System.out.println( "\nCoeff around 0.01, " +
      "homogeneous population.");
  System.out.println( "Coeff around 0.1, " +
      " +/- homogeneous population.");
  System.out.println( "Coeff around 0.5, " +
      "  heterogenous population.");
}


public static void main(String[] args)
 {
   System.out.println( "MEAN AND VAR OF A LIST OF DATA");
   //Example 19 page 9.
   double  DataVect[]  = {5, 3, 4.5, 2};
   meanVarVector(DataVect);
 }

}
```

27, pag. 18. The program has two versions of the method *sumVector* which sums the entries of a given vector, one for the original vector and the other for the vector with its squares. One can see that apart from minor but transcendental changes the code is repeated, so it amounts to a duplication. At the end this is a form of reuse that functions in spite of being primitive. Now, is being primitive a defect or a virtue? It depends on the context. In computing science of high level languages, such as Java, it is a serious pitfall but for languages of low level, such as assembler, being primitive is the only possible way of life. By the way, the genome has a very primitive style of design, so much that it has been argued that its complexity arouse as the natural fruit of evolution through duplications. On the other hand, the designers of programming languages noticed that reuse, or evolution, can be greatly facilitated if one enables each method to be a function, i.e., a machine that processes in a unique way a given input into a unique output that must be clearly separated from the input.

30, pag. 19. Mean and variance according to the functional style:

```
//Program F30 meanVarList4
/* This program computes the mean and
   variance of a list of data.
   Functional style of design:
   1) Code is compartmentalized in methods.
   2) Every method is a function with
      a clearly specified input and
      with an output that is ready for reuse
      in other methods.
   3) No global variables.
   4) Methods  are ready for reuse in other classes.
   */

public class meanVarList4
{

 /*
  * No global variables
  */


 //Data are summed.
 //The input is a Vector, the output is of type double.
 private static double sumVector(double[] V)
 {
  double sum = 0;
  for(int i = 0; i < V.length; i++)
sum  = sum +  V[i];
  return sum;
 }



//The mean of a list of data
private static double meanVector(double[] V)
 {
  int n = V.length;
  double mean =  sumVector(V) / n;
  return mean;
 }



//Every entry of a Vector is squared.
```

```
//The input is a Vector, the output is also a Vector.
private static double[] squareVector(double[] V)
{
//Zeroed initialization of a Vector
double[]  V2 = new double[V.length];
//The square of each entry of F is calculated
//and kept in list.F
for(int i = 0; i < V.length; i++)
V2[i] = V[i] * V[i];
return V2;
}

//Documentation is printed
private static void explanation()
{
  System.out.println( "\nCoeff around 0.01, " +
                      "homogeneous population." +
                      "\nCoeff around 0.1, " +
                      " +/- homogeneous population."+
                      "\nCoeff around 0.5, " +
                      "  heterogenous population.");
 }


//This is the core of the main class
//Here, commands are given to execute the overall task.
private static void meanVarVector(double[] V)
{
 String a = "x";
 String b = "x2 = x^2";
 System.out.printf("%n %10s  %11s", a, b);
 int n = V.length;
 double[] squareVect = new double[n];
 squareVect = squareVector(V);
 for(int i = 0; i < n; i++ )
    System.out.printf("%n %10.3f   %10.3f ",
       V[i], squareVect[i] );
 double sumx = sumVector(V);
 double mean = meanVector(V);
 double sumx2 = sumVector(squareVect);
 System.out.printf("%n Sum %6.3f   %10.3f ",
                sumx,        sumx2 );
 System.out.print("\n \n");
 System.out.println("n = number of data = " + n);

 System.out.println("\nsumx = " + sumx);
```

```
 System.out.println("Mean = sumx / n");
 System.out.println("Mean = " + mean);
 System.out.println("sumx2 = " + sumx2);
 System.out.println("variance = " +
   "(n * sumx2 - sumx*sumx)/ (n*( n-1))");
 double var = (n * sumx2 - sumx*sumx)/ (n*( n-1));
 System.out.println( "Variance = "+ var);
 double deviation = Math.pow( var, 0.5);
 System.out.println( "Deviation = "+ deviation);
 double coeffVar =  deviation/mean;
 System.out.println( "Coefficient of variation " +
                 "= deviation/mean");
 System.out.println( "Coefficient of variation = "
                                  + coeffVar);
 explanation();
}

//Input is read and directed to processing methods
public static void main(String[] args)
{
  System.out.println( "MEAN AND VAR OF A LIST OF DATA");
  //Example 19 page 9.
  double  DataVect[]  = {5, 3, 4.5, 2};
  meanVarVector(DataVect);
}

}
```

37, pag. 27. A tale according to a gossipless, transparent, functional style of design.

```
//Program F37 meanVarList7
/* This program computes the mean and
   variance of a list of data.
   Gossipless, transparent, functional style of design:
   1) Code is compartmentalized in methods.
   2) Every method is a function with
      a clearly specified input and
      with an output that is ready for reuse
      in other methods.
   3) No global variables.
   4) Methods  are ready for reuse in other classes.
   5) The code, its output, the problem and its solution
      are all made on the same image.
```

```
   6) A table is defined by an inner class
      as a personalized object.
   7) Domestic problems are solve at own house.
   */

public class meanVarList7
{

 /*
  * No global variables
  */

 //The outer application is instantiated
 private static meanVarList7 p = new meanVarList7();


 //======== Inner class definition=======

 //This inner class defines a new type: Table.
 //It converts array F (two dimensional) into
 //a class or object, ready for reuse.
 private  class Table
 {
  //The class Table fundamentally has three elements:
  //a matrix with two columns and its length,
  //which is the number of rows and
  //the titles of columns.
  int length;
  String[] Titles = { "x",  "x2 = x^2"};
  double M[][] = new double[2000][2];

  //The input of Table is defined here
  //by means of an initialization procedure
  //that is called constructor.
  //The natural input is the raw data,
  //which is consigned in a one dimensional array.
  Table(double[] V )
  {
   length = V.length;
   //The matrix M of Table is initialized
   //Raw data go in the first column.
   //Squares in the second.
   for(int i = 0; i < length; i++)
   {
 M[i][0] = V[i];
 M[i][1] = V[i]*V[i];
```

```
   }
  }

  //Data of column k are summed.
  //The input is a table, the output is of type double.
  private  double sum(Table  T, int k)
  {
   double sum = 0;
   for(int i = 0; i < T.length; i++)
  sum  = sum +  T.M[i][k];
   return sum;
  }

 private  void printTable(Table T)
  {
    System.out.printf("%n %10s  %11s", T.Titles [0], T.Titles [1]);
 for ( int i = 0; i < T.length; i++ )
    System.out.printf("%n %10.3f   %10.3f ",
        T.M[i][0], T.M[i][1] );
 System.out.println( );
  }
}//end of inner class

//=== Methods of the outer class meanVarList7 ===



//Documentation is printed
private static void explanation()
{
  System.out.println( "\nCoeff around 0.01, " +
                      "homogeneous population." +
                      "\nCoeff around 0.1, " +
                      " +/- homogeneous population."+
                      "\nCoeff around 0.5, " +
                      "  heterogenous population.");
 }


//This is the core of the main class
//Here, commands are given to execute the overall task.
private static void meanVarVector(double[] V)
{
 //A table is declared
 Table T =  p.new   Table(V);
 T.printTable(T);
```

```
 double sumx = T.sum(T,0);
 int n = T.length;
 double mean = sumx/ n;
 double sumx2 = T.sum(T,1);
 System.out.printf("%n Sum %6.3f   %10.3f ",
                sumx,        sumx2 );
 System.out.print("\n \n");
 System.out.println("n = number of data = " + n);

 System.out.println("\nsumx = " + sumx);
 System.out.println("Mean = sumx / n");
 System.out.println("Mean = " + mean);
 System.out.println("sumx2 = " + sumx2);
 System.out.println("variance = " +
   "(n * sumx2 - sumx*sumx)/ (n*( n-1))");
 double var = (n * sumx2 - sumx*sumx)/ (n*( n-1));
 System.out.println( "Variance = "+ var);
 double deviation = Math.pow( var, 0.5);
 System.out.println( "Deviation = "+ deviation);
 double coeffVar =  deviation/mean;
 System.out.println( "Coefficient of variation " +
                 "= deviation/mean");
 System.out.println( "Coefficient of variation = "
                                  + coeffVar);
 explanation();
}

//Input is read and directed to processing methods
public static void main(String[] args)
{
  System.out.println( "MEAN AND VAR OF A LIST OF DATA");
  //Example 19 page 9.
  double  DataVect[]  = {5, 3, 4.5, 2};
  meanVarVector(DataVect);
}

}
```

39, pag. 28. Mean and variance of a frequency table. OOP style:

```
//Program F39 meanFreqTable
//Finds the mean and variance of a table
//of absolute frequencies.
//Uses the method of the Sxx.
```

```
//Gossipless, transparent, functional OOP style.
//Gossipless means that
//domestic problems are solve at home.
public class meanFreqTable
{

  /*
   * No global variables
   */

  //The outer application is instantiated
  private static meanFreqTable p = new meanFreqTable();


  //======== Inner class definition=======

  //This inner class defines a new type: Table.
  //It converts array F (two dimensional) into
  //a class or object, ready for reuse.
  private  class Table
   {
//The class Table has two main elements:
//1) The titles of  the columns, whose length
//gives the number of columns.
//2) A matrix, whose length is the number of rows.

String[] Titles = { "x",  "F",   "xF", "x2F"};
double M[][] = new double[2000][Titles.length];
int nRows;
//The input of Table is defined here
//by means of an initialization procedure
//that is called constructor.
//The natural input is the frequency table
//which is consigned in a matrix FT with 2 columns.
Table(double[][] FT )
 {
   nRows = FT.length;
   //The matrix M of Table is initialized
   //Raw data go in   columns 0 and 1.
   //Squares in the second.
   for(int i = 0; i < nRows; i++)
   {
 M[i][0] = FT[i][0];
 M[i][1] = FT[i][1];
 M[i][2] = M[i][0] * M[i][1];
 M[i][3] = M[i][0] * M[i][2];
```

```
   }
  }

//Data of column k are summed.
//The input is a table, the output is of type double.
private  double sum(Table  T, int k)
 {
  double sum = 0;
  for(int i = 0; i < T.nRows; i++)
  sum  = sum +  T.M[i][k];
  return sum;
 }

private  void printTable(Table T)
 {
  System.out.printf("%n %10s %10s %10s %10s",
            T.Titles [0], T.Titles [1],
             T.Titles [2],T.Titles [3]);
      for ( int i = 0; i < T.nRows; i++ )
 System.out.printf(
"%n %10.3f %10.3f %10.3f %10.3f ",
T.M[i][0], T.M[i][1], T.M[i][2], T.M[i][3] );
  System.out.println( );
 }
  }//end of inner class


//=== Methods of the outer class meanFreqTable  ===


//Documentation is printed
private static void explanation()
 {
  System.out.println( "\nCoeff around 0.01, " +
                        "homogeneous population." +
                        "\nCoeff around 0.1, " +
                        " +/- homogeneous population."+
                        "\nCoeff around 0.5, " +
                        "  heterogenous population.");
 }

 // Mean and variance of a frequency table
 private static void meanVarTable(double FreqTable[][])
 {
  //A table is declared
  Table T =  p.new   Table(FreqTable);
```

```
  T.printTable(T);
  double n = T.sum(T,1);
  double sumxF = T.sum(T,2);
  double sumx2F = T.sum(T,3);
  System.out.printf("Sum %18.3f %10.3f %10.3f ",
             n,   sumxF,        sumx2F );
  double mean = sumxF/ n;
  System.out.print(
  "\n\n n = Number of  objects = sum F = "+ (int) n);
  System.out.printf( "\n sum xF  = %10.3f ", sumxF);
  System.out.printf( "\n sum x2F  = %10.3f ",  sumx2F);
  System.out.printf( "\n Mean = sum xF / n = %10.3f ", mean);
  double Sxx  = sumx2F -    Math.pow(sumxF, 2)/n;
  System.out.printf(
 "\n Sxx = sum x2F - (sumxF)^2 /n = %10.3f ", Sxx);
  double var = Sxx / (n-1);
  System.out.printf(
 "\n Variance =  Sxx/(n-1) = %10.3f ", var);
  double deviation = Math.sqrt(var);
  System.out.printf(
 "\n Deviation =  %10.3f ", deviation);
  double c = deviation/ mean;
  System.out.printf(
     "\n Coefficient of variation =  %10.3f \n", c);
  explanation();
 }


public static void main(String[] args)
{
 System.out.println("MEAN-VAR OF A FREQUENCY TABLE");
 //Frequency table: first coordinate = class marker
 //second coordinate = frequency

 //Example 20, pag 10
 double  FreqT[][] =
        {{4, 3},
         {7, 6},
         {10, 10},
         {13, 15},
         {16, 4},
         {19, 2}};
  meanVarTable(FreqT);
}

}//end of class
```

41, pag. 28. Mean and variance of a cumulative frequency table:

```
//Program F41 meanCumTable
//Finds the mean and variance of a table
//of cumulative frequencies.
//Uses the method of the Sxx.
//Style: reuse.
public class meanCumTable
{

  /*
   * No global variables
   */

  //The outer application is instantiated
  private static meanCumTable p = new meanCumTable();


  //======== Inner class definition=======

  //This inner class defines a new type: Table.
  //It converts array F (two dimensional) into
  //a class or object, ready for reuse.
  private  class Table
   {
//The class Table has two main elements:
//1) The titles of  the columns, whose length
//gives the number of columns.
//2) A matrix, whose length is the number of rows.

String[] Titles = { "x",  "F",   "xF", "x2F"};
double M[][] = new double[2000][Titles.length];
int nRows;
//The input of Table is defined here
//by means of an initialization procedure
//that is called constructor.
//The natural input is the frequency table
//which is consigned in a matrix FT with 2 columns.
Table(double[][] FT )
 {
   nRows = FT.length;
   //The matrix M of Table is initialized
   //Raw data go in   columns 0 and 1.
   //Squares in the second.
   for(int i = 0; i < nRows; i++)
```

```
   {
 M[i][0] = FT[i][0];
 M[i][1] = FT[i][1];
 M[i][2] = M[i][0] * M[i][1];
 M[i][3] = M[i][0] * M[i][2];
   }
  }

//Data of column k are summed.
//The input is a table, the output is of type double.
private  double sum(Table  T, int k)
 {
  double sum = 0;
  for(int i = 0; i < T.nRows; i++)
  sum  = sum +  T.M[i][k];
  return sum;
 }

private  void printTable(Table T)
 {
  System.out.printf("%n %10s %10s %10s %10s",
            T.Titles [0], T.Titles [1],
             T.Titles [2],T.Titles [3]);
      for ( int i = 0; i < T.nRows; i++ )
 System.out.printf(
"%n %10.3f %10.3f %10.3f %10.3f ",
T.M[i][0], T.M[i][1], T.M[i][2], T.M[i][3] );
  System.out.println( );
 }
  }//end of inner class


//=== Methods of the outer class meanCumTable  ===


//Documentation is printed
private static void explanation()
 {
  System.out.println( "\nCoeff around 0.01, " +
                        "homogeneous population." +
                        "\nCoeff around 0.1, " +
                        " +/- homogeneous population."+
                        "\nCoeff around 0.5, " +
                        "  heterogenous population.");
 }
```

```
 // Mean and variance of a frequency table
 private static void meanVarTable(double FreqTable[][])
 {
  //A table is declared
  Table T =  p.new   Table(FreqTable);
  T.printTable(T);
  double n = T.sum(T,1);
  double sumxF = T.sum(T,2);
  double sumx2F = T.sum(T,3);
  System.out.printf("Sum %18.3f %10.3f %10.3f ",
             n,   sumxF,        sumx2F );
  double mean = sumxF/ n;
  System.out.print(
  "\n\n n = Number of  objects = sum F = "+ (int) n);
  System.out.printf( "\n sum xF  = %10.3f ", sumxF);
  System.out.printf( "\n sum x2F  = %10.3f ",  sumx2F);
  System.out.printf( "\n Mean = sum xF / n = %10.3f ",
             mean);
  double Sxx  = sumx2F -    Math.pow(sumxF, 2)/n;
  System.out.printf(
 "\n Sxx = sum x2F - (sumxF)^2 /n = %10.3f ", Sxx);
  double var = Sxx / (n-1);
  System.out.printf(
 "\n Variance =  Sxx/(n-1) = %10.3f ", var);
  double deviation = Math.sqrt(var);
  System.out.printf(
 "\n Deviation =  %10.3f ", deviation);
  double c = deviation/ mean;
  System.out.printf(
     "\n Coefficient of variation =  %10.3f \n", c);
  explanation();
 }

private static void meanVarCumTable(double[][] cumFreqT)
{
 System.out.println(
 "\nCUMULATIVE FREQUENCY TABLE");
 for ( int i = 0; i < cumFreqT.length; i++ )
 System.out.println( cumFreqT[ i ][0]+
        "\t "+ cumFreqT[ i ][1]);
 //Absolute frequencies are calculated
 //and kept in FreqT, a table with two columns.
 double FreqT[][] = new double[cumFreqT.length][2];

 FreqT[0] = cumFreqT[0];
 int lx = cumFreqT.length;
```

```
 for(int i = 1; i < lx;  i++)
 {
  FreqT[i][0] = cumFreqT[i][0];
  FreqT[i][1] = cumFreqT[i][1]-cumFreqT[i-1][1];
 }
 System.out.println(
 "\n MEAN-VAR OF ASSOCIATED  FREQUENCY TABLE");
  meanVarTable(FreqT);//reuse
 }

public static void main(String[] args)
{
 System.out.println(
 "MEAN-VAR OF A CUMULATIVE FREQUENCY TABLE");
 //Cumulative table: first coordinate = class marker
 //second coordinate = cumulative frequency

 //Problem 1 of version 1,  pag 42.
 double  cumFreqT[][] =
        {{32,7},
         {40,12},
         {45,17},
         {50,45},
         {55,52},
         {60,58}};
  meanVarCumTable(cumFreqT);
}

}//end of class
```

43, pag. 33. The style is reuse because the program reuses a previous code developed to calculate frequency tables. The style is not transparent because the program uses diverse objects to make the task while the spirit of the program sees only one object: an extended table with the borders of the intervals, class markers, and the necessary columns to calculate the variance. The same problem appears recurrently everywhere, and because we identify evolution in first instance with reuse, we see that evolution and good style do not go hand in hand. This observation allows to imagine a test to decide whether or not a given software was devised by evolution else by refined stylistic intelligent developers.

44, pag. 33. Mean and variance of a frequency table, variant.

```
//Program F44 ListToFT2
//Groups data in interval classes.
```

```
//Outputs a frequency table with its
//mean and variance.
//Input: a list of data +
//inferior border and length of the intervals.
//Style = reuse.

public class ListToFT2
{
 //The outer application is instantiated
 private static ListToFT2 p = new ListToFT2();


 //======== Inner class definition=======

 //This inner class defines a new type: Table.
 //It converts array F (two dimensional) into
 //a class or object, ready for reuse.
 private  class Table
  {
   //The class Table has two main elements:
   //1) The titles of  the columns, whose length
   //gives the number of columns.
   //2) A matrix, whose length is the number of rows.

   String[] Titles = { "x",  "F",   "xF", "x2F"};
   double M[][] = new double[2000][Titles.length];
   int nRows;
   //The input of Table is defined here
   //by means of an initialization procedure
   //that is called constructor.
   //The natural input is the frequency table
   //which is consigned in a matrix FT with 2 columns.
   Table(double[][] FT )
  {
   nRows = FT.length;
   //The matrix M of Table is initialized
   //Raw data go in   columns 0 and 1.
   //Squares in the second.
   for(int i = 0; i < nRows; i++)
    {
 M[i][0] = FT[i][0];
 M[i][1] = FT[i][1];
 M[i][2] = M[i][0] * M[i][1];
 M[i][3] = M[i][0] * M[i][2];
}
   }
```

```
  //Data of column k are summed.
  //The input is a table, the output is of type double.
  private  double sum(Table  T, int k)
{
   double sum = 0;
   for(int i = 0; i < T.nRows; i++)
  sum  = sum +  T.M[i][k];
   return sum;
}

  private  void printTable(Table T)
   {
    System.out.printf("%n %10s %10s %10s %10s",
      T.Titles [0], T.Titles [1],
      T.Titles [2],T.Titles [3]);
    for ( int i = 0; i < T.nRows; i++ )
    System.out.printf(
"%n %10.3f %10.3f %10.3f %10.3f ",
T.M[i][0], T.M[i][1], T.M[i][2], T.M[i][3] );
System.out.println( );
   }
  }//end of inner class


 //=== Methods of the outer class ListToFT  ===


 private static void printList(double DataVector[],
                               String s)
 {
  System.out.println( "\n " +s );
  int nData = DataVector.length;

  boolean go = true;
  int nCols = 10;
  int nRows = 9;
  int counter = 0;
  for (int i = 0; (i < nRows) & go; i++)
{
 for (int j = 0; (j < nCols) & go; j++)
{
 System.out.print( DataVector[i*10 + j] + "   ");
 if (counter >= nData-1) go = false;
  counter = counter+1;
    }
```

```
      System.out.println( );
 }
   System.out.print("\nNumber of data = " +nData + "\n");
  }

 //List is the input, a frequency table is the output
 private static double[][] group(double List[],
        double Borders[], double classMarkers[]  )
 {
  int nClasses = classMarkers.length;
  double FreqTable[][] = new double[nClasses][2];
  int nData = List.length;

  for(int j=0; j < nClasses; j++)
   {
FreqTable[j][0] = classMarkers[j];
FreqTable[j][1] = 0;
   }
  for(int i=0; i< nData; i++)
   {
    for(int j=0; j < nClasses; j++)
 {
  if((Borders[j] <= List[i]) & (List[i] < Borders[j+1]))
  FreqTable[j][1] =FreqTable[j][1] +1;
}

 }

  return FreqTable;
 }


 //Documentation is printed
 private static void explanation()
  {
   System.out.println( "\nCoeff around 0.01, " +
                   "homogeneous population." +
                   "\nCoeff around 0.1, " +
                   " +/- homogeneous population."+
                   "\nCoeff around 0.5, " +
                   "  heterogenous population.");
  }


// Mean and variance of a frequency table
 private static void meanVarTable(double FreqTable[][])
```

```
  {
   //A table is declared
   Table T =  p.new   Table(FreqTable);
   T.printTable(T);
   double n = T.sum(T,1);
   double sumxF = T.sum(T,2);
   double sumx2F = T.sum(T,3);
   System.out.printf("Sum %18.3f %10.3f %10.3f ",
      n,   sumxF,        sumx2F );
   double mean = sumxF/ n;
   System.out.print(
  "\n\n n = Number of  objects = sum F = "+ (int) n);
   System.out.printf( "\n sum xF  = %10.3f ", sumxF);
   System.out.printf( "\n sum x2F  = %10.3f ",  sumx2F);
   System.out.printf( "\n Mean = sum xF / n = %10.3f ",
             mean);
   double Sxx  = sumx2F -    Math.pow(sumxF, 2)/n;
   System.out.printf(
 "\n Sxx = sum x2F - (sumxF)^2 /n = %10.3f ", Sxx);
   double var = Sxx / (n-1);
   System.out.printf(
 "\n Variance =  Sxx/(n-1) = %10.3f ", var);
   double deviation = Math.sqrt(var);
   System.out.printf(
 "\n Deviation =  %10.3f ", deviation);
   double c = deviation/ mean;
   System.out.printf(
    "\n Coefficient of variation =  %10.3f \n", c);
   explanation();
  }

 //Prints data in vector
 private static void printVector(double Vect[], int n)
  {
   //System.out.println( "Data are" );
   for ( int i = 0; i < n; i++ )
   System.out.println( Vect[ i ]  );
  }

 //The borders and class marker of each interval
 //is calculated. Data is grouped into a frequency table.
 //final output = Mean and var of the freq table.
 private static void work(double List[],
double infLimit, double intervalLength)
 {
   //Calculate minimal value of data
```

```
   double min = List[0];
   for(int j=1; j < List.length; j++)
    if (List[j] < min)  min = List[j];
   //Calculate maximal value of data
   double max = List[0];
   for(int j=1; j < List.length; j++)
    if (List[j] > max)  max = List[j];
   System.out.println( "\nMax value = " + max);
   //Calculates number of classes
   int nClasses  = (int) ((max- infLimit)/intervalLength)+1;
   System.out.println( "Number of classes = " + nClasses);
   //Calculates interval borders and class markers
   //Grouping mode 2: defines inferior limit and length
   //of intervals.
   double classMarkers[] = new double[nClasses];
   double  Borders[] = new double[nClasses+2];
   Borders[0] = infLimit;
   classMarkers[0] = infLimit + intervalLength/2;
   for(int j=1; j < nClasses; j++)
     classMarkers[j] = classMarkers[j-1] + intervalLength;
   for(int j=1; j <= nClasses+1; j++)
     Borders[j] = Borders[j-1] + intervalLength;
   System.out.println( "\nBorders are" );
   printVector(Borders,nClasses+1 );
   System.out.println( "\nClass markers are" );
   printVector(classMarkers,nClasses );
  double FreqTable[][] = new double[nClasses][2];
  System.out.println(
 "\nEXTENDED TABLE OF ABSOLUTE FREQUENCIES" );
  //reuse
  FreqTable = group(List, Borders, classMarkers );
  meanVarTable(FreqTable);
 }

public static void main(String[] args)
 {
  //Declaration of data:
  //observed values as a list
  //Example 24 pag 12
  double    List[] =
{
48.5, 49.2, 51.0, 50.9, 48.7, 50.5, 49.5, 50.8, 50.0, 51.1,
50.7, 51.1, 49.3, 49.1, 50.0, 48.7, 51.2, 49.2, 49.0, 49.3,
49.1, 48.7, 48.6, 49.2, 49.9, 50.1, 50.1, 50.9, 52.4, 50.2,
49.5, 50.9, 51.4, 49.7, 49.8, 50.8, 50.3, 51.8, 50.0, 51.1,
51.3, 50.1, 50.4, 51.1, 49.8, 49.8, 50.1, 50.7, 50.1, 49.9,
```

```
49.7, 51.1, 49.7, 49.9, 49.8, 50.5, 49.7, 50.8, 50.7, 50.7,
50.8,  51.5, 51.0
    };
  System.out.println("FROM DATA LIST TO FREQUENCY TABLE:");
  printList(List, "Raw data");

  //Calculate minimal value of data
  double min = List[0];
  for(int j=1; j < List.length; j++)
    if (List[j] < min)  min = List[j];
  System.out.println( "\nMin value = " + min);


  double infLimit = min;
  //Interval length
  double intervalLength = 1;
  System.out.println("Inferior limit = " + infLimit);
  System.out.println("Interval length = "
  + intervalLength);
  work(List, infLimit, intervalLength);
 }
}//end of main classes
```

52, pag. 48. This is toolBox4, a program to process the basic statistics of quantitative data.

```
//Program F52 toolBox4
/*
 This is a tool box of methods to calculate
 some statistics of a list of data.

 How to use this program:

 1. Append it to any project and run it.
 2. This program is a tool box because its main method
    can be filled with any allowed method.
 3. Most relevant allowed methods are listed in the main
    but only one is activated, all others are
    muted.
 4. To silence one method, append // as a prefix.
    To activate a method, delete the prefix //.
 5. For a given task, choose the best suited method
    among the diverse variants.
 6. To run a method with your own data,
    modify the inbuilt data according to your needs.
 7. This first version has just one aim:
```

```
    To implement fundamental tasks as transparently
    as possible. So, it looks rather like an array of
    independent, non interacting  methods.
 8. To inquire about unknown terms, drag the mouse over
    the term and wait until Eclipse displays the
    corresponding documentation.
 9. To go   from the name of a
    method  to its implementation:
     Pose the cursor at the end of the line
     with the name of the method and press the space bar,
     next press F3: you will be transported to the
     implementation of the method.
 10. To mark a specific site in the program:
     -Activate line numbering.
     -Pose the cursor over the line you want to mark.
     -Go to the corresponding number at the left margin
      of the Editor Window and right click.
     -Choose the Add Bookmark menu and punch it.
     -A blue marker will appear in the right margin of the
      Editor Window. By clicking over it, one can
      return to the desired place.
      One can also play with the menu
      Window -> ShowView -> Outline
      A list with all methods will appear:
      click over any one and you will
      transported to the corresponding place
      along the text of the program.

*/



public class toolBox4
{

 //The outer application is instantiated
 private static toolBox4 p = new toolBox4();




//==============================
//MAIN--MAIN--MAIN--MAIN--MAIN
//==============================

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

```
  {
   meanVarList();
   meanVarFreqTable();
   meanVarCumTable();
   groupData();
  }




//=============================================
//               IMPLEMENTATIONS
//=============================================




//=============================================
//      Methods of general use
//=============================================


//Prints a list and its title
private static void printList(double DataVector[],
                              String title)
{
System.out.println( "\n " + title );
int nData = DataVector.length;
boolean go = true;
int nCols = 10;
int nRows = 9;
int counter = 0;
for (int i = 0; (i < nRows) & go; i++)
 {
  for (int j = 0; (j < nCols) & go; j++)
   {
    System.out.print( DataVector[i*10 + j] + "   ");
    if (counter >= nData-1) go = false;
    counter = counter+1;
   }
 System.out.println( );
}
System.out.print("\nNumber of data = " +nData + "\n");
}
```

```
//Prints n data in vector
private static void printVector(double Vect[], int n)
{
//System.out.println( "Data are" );
for ( int i = 0; i < n; i++ )
   System.out.println( Vect[ i ]  );
}


//Prints  a matrix and is title
private static void printMatrix(double Matrix[][],
                        String title)
{
System.out.println( title );
int m = Matrix.length;
 for ( int i = 0; i < m; i++ )
 System.out.println( Matrix[ i ][0]+
        "\t "+ Matrix[ i ][1]);
}


//Documentation is printed
private static void explanation()
{
 System.out.println( "\nCoeff around 0.01, " +
                     "homogeneous population." +
                     "\nCoeff around 0.1, " +
                     " +/- homogeneous population."+
                     "\nCoeff around 0.5, " +
                     "  heterogenous population.");
}


//=============================================
//       MEAN AND VAR O A LIST OF DATA
//=============================================



 //======== Inner class definition=======

 //This inner class defines a new type: Table.
 //It converts array F (two dimensional) into
 //a class or object, ready for reuse.
 private  class Table
```

```
{
 //The class Table fundamentally has three elements:
 //a matrix with two columns and its length,
 //which is the number of rows and
 //the titles of columns.
 int length;
 String[] Titles = { "x",  "x2 = x^2"};
 double M[][] = new double[2000][2];

 //The input of Table is defined here
 //by means of an initialization procedure
 //that is called constructor.
 //The natural input is the raw data,
 //which is consigned in a one dimensional array.
 Table(double[] V )
 {
  length = V.length;
  //The matrix M of Table is initialized
  //Raw data go in the first column.
  //Squares in the second.
  for(int i = 0; i < length; i++)
  {
M[i][0] = V[i];
M[i][1] = V[i]*V[i];
  }
 }

 //Data of column k are summed.
 //The input is a table, the output is of type double.
 private  double sum(Table  T, int k)
 {
  double sum = 0;
  for(int i = 0; i < T.length; i++)
 sum  = sum +  T.M[i][k];
  return sum;
 }

private  void printTable(Table T)
 {

   System.out.printf("%n %10s  %11s", T.Titles [0], T.Titles [1]);
for ( int i = 0; i < T.length; i++ )
   System.out.printf("%n %10.3f   %10.3f ",
       T.M[i][0], T.M[i][1] );
System.out.println( );
 }
```

```
}//end of inner class


//=== Methods of the outer class toolBox4 ===


//This is the core of the main class
//Here, commands are given to execute the overall task.
private static void meanVarVector(double[] V)
{
 //A table is declared
 Table T =  p.new   Table(V);
 T.printTable(T);
 double sumx = T.sum(T,0);
 int n = T.length;
 double mean = sumx/ n;
 double sumx2 = T.sum(T,1);
 System.out.printf("%n Sum %6.3f   %10.3f ",
                sumx,        sumx2 );
 System.out.print("\n \n");
 System.out.println("n = number of data = " + n);

 System.out.println("\nsumx = " + sumx);
 System.out.println("Mean = sumx / n");
 System.out.println("Mean = " + mean);
 System.out.println("sumx2 = " + sumx2);
 System.out.println("variance = " +
   "(n * sumx2 - sumx*sumx)/ (n*( n-1))");
 double var = (n * sumx2 - sumx*sumx)/ (n*( n-1));
 System.out.println( "Variance = "+ var);
 double deviation = Math.pow( var, 0.5);
 System.out.println( "Deviation = "+ deviation);
 double coeffVar =  deviation/mean;
 System.out.println( "Coefficient of variation " +
                 "= deviation/mean");
 System.out.println( "Coefficient of variation = "
                                 + coeffVar);
 explanation();
}

//A list of data  is  processed
private static void meanVarList( )
{
  System.out.println("\n\n");
  System.out.println( "MEAN AND VAR OF A LIST OF DATA");
  //Example 19 page 9.
```

```
  double  List[]  = {5, 3, 4.5, 2};
  //Prints list and title
  printList(List,  "Raw data");
  System.out.println( "\nTABLE WITH x AND ITS SQUARE");
  meanVarVector(List);
}

//=================================================
// MEAN AND VAR OF A TABLE OF ABSOLUTE FREQUENCIES
//=================================================


//======== Inner class definition=======

//This inner class defines a new type: Table.
//It converts array F (two dimensional) into
//a class or object, ready for reuse.
private  class TableFreq
 {
//The class Table has two main elements:
//1) The titles of  the columns, whose length
//gives the number of columns.
//2) A matrix, whose length is the number of rows.

String[] Titles = { "x",  "F",   "xF", "x2F"};
double M[][] = new double[2000][Titles.length];
int nRows;
//The input of Table is defined here
//by means of an initialization procedure
//that is called constructor.
//The natural input is the frequency table
//which is consigned in a matrix FT with 2 columns.
TableFreq(double[][] FT )
 {
   nRows = FT.length;
   //The matrix M of Table is initialized
   //Raw data go in   columns 0 and 1.
   //Squares in the second.
   for(int i = 0; i < nRows; i++)
   {
 M[i][0] = FT[i][0];
 M[i][1] = FT[i][1];
 M[i][2] = M[i][0] * M[i][1];
 M[i][3] = M[i][0] * M[i][2];
   }
  }
```

```
//Data of column k are summed.
//The input is a table, the output is of type double.
private  double sum(TableFreq  T, int k)
 {
  double sum = 0;
  for(int i = 0; i < T.nRows; i++)
  sum  = sum +  T.M[i][k];
  return sum;
 }

private  void printTable(TableFreq T)
 {
  System.out.println("\nExtended matrix to " +
                "calculate variance:");
  System.out.printf("%n %10s %10s %10s %10s",
            T.Titles [0], T.Titles [1],
             T.Titles [2],T.Titles [3]);
    for ( int i = 0; i < T.nRows; i++ )
 System.out.printf(
"%n %10.3f %10.3f %10.3f %10.3f ",
T.M[i][0], T.M[i][1], T.M[i][2], T.M[i][3] );
  System.out.println( );
 }
}//end of inner class


//=== Methods of the outer class toolBox4  ===



// Mean and variance of a frequency table
private static void meanVarFreqTable(double FreqTable[][])
{
//A table is declared
TableFreq T =  p.new   TableFreq(FreqTable);
T.printTable(T);
double n = T.sum(T,1);
double sumxF = T.sum(T,2);
double sumx2F = T.sum(T,3);
System.out.printf("Sum %18.3f %10.3f %10.3f ",
             n,   sumxF,        sumx2F );
double mean = sumxF/ n;
System.out.print(
  "\n\n n = Number of  objects = sum F = "+ (int) n);
System.out.printf( "\n sum xF  = %10.3f ", sumxF);
```

```
System.out.printf( "\n sum x2F  = %10.3f ",  sumx2F);
System.out.printf( "\n Mean = sum xF / n = %10.3f ", mean);
double Sxx  = sumx2F -    Math.pow(sumxF, 2)/n;
System.out.printf(
 "\n Sxx = sum x2F - (sumxF)^2 /n = %10.3f ", Sxx);
double var = Sxx / (n-1);
System.out.printf(
 "\n Variance =  Sxx/(n-1) = %10.3f ", var);
double deviation = Math.sqrt(var);
System.out.printf(
 "\n Deviation =  %10.3f ", deviation);
double c = deviation/ mean;
System.out.printf(
   "\n Coefficient of variation =  %10.3f \n", c);
explanation();
}


//A Frequency Table is processed
private static void meanVarFreqTable()
{
System.out.println("\n\n");
System.out.println("\n\nMEAN-VAR OF A FREQUENCY TABLE");
//Frequency table: first coordinate = class marker
//second coordinate = frequency

//Example 20, pag 10
double  FreqT[][] =
        {{4, 3},
         {7, 6},
         {10, 10},
         {13, 15},
         {16, 4},
         {19, 2}};

printMatrix(FreqT, "\n Raw data");
meanVarFreqTable(FreqT);
}


//=================================================
//MEAN AND VAR OF A TABLE OF CUMULATIVE FREQUENCIES
//=================================================


private static void meanVarCumTable(double[][] cumFreqT)
```

```
{
 printMatrix(cumFreqT, "\n Raw data");
 //Absolute frequencies are calculated
 //and kept in FreqT, a matrix with two columns.
 double FreqT[][] = new double[cumFreqT.length][2];

 FreqT[0] = cumFreqT[0];
 int lx = cumFreqT.length;
 for(int i = 1; i < lx;  i++)
 {
  FreqT[i][0] = cumFreqT[i][0];
  FreqT[i][1] = cumFreqT[i][1]-cumFreqT[i-1][1];
 }
 System.out.println("\n\n");
 System.out.println(
 "\n MEAN-VAR OF ASSOCIATED  FREQUENCY TABLE");
  meanVarFreqTable(FreqT);//reuse
 }

//A cumulative frequency table is processed
public static void meanVarCumTable()
{
 System.out.println(
 "MEAN-VAR OF A CUMULATIVE FREQUENCY TABLE");
 //Cumulative table: first coordinate = class marker
 //second coordinate = cumulative frequency

 //Problem 1 of version 1,  pag 42.
 double  cumFreqT[][] =
        {{32,7},
         {40,12},
         {45,17},
         {50,45},
         {55,52},
         {60,58}};
  meanVarCumTable(cumFreqT);
}


//=================================================
// GROUPING OF DATA IN INTERVALS + MEAN & VAR
//=================================================



//List is the input, a frequency table is the output
```

```
private static double[][] group(double List[],
        double Borders[], double classMarkers[]  )
{
 int nClasses = classMarkers.length;
 double FreqTable[][] = new double[nClasses][2];
 int nData = List.length;

 for(int j=0; j < nClasses; j++)
  {
FreqTable[j][0] = classMarkers[j];
FreqTable[j][1] = 0;
  }
 for(int i=0; i< nData; i++)
  {
   for(int j=0; j < nClasses; j++)
 {
  if((Borders[j] <= List[i]) & (List[i] < Borders[j+1]))
  FreqTable[j][1] =FreqTable[j][1] +1;
}

 }

 return FreqTable;
}
```

```
//Data is grouped into a frequency table.
//The borders and class marker of each interval
//is calculated.
//Final output = Mean and var of the freq table.
private static void group(double List[],
double infLimit, double intervalLength)
{
  //Calculate minimal value of data
  double min = List[0];
  for(int j=1; j < List.length; j++)
   if (List[j] < min)  min = List[j];
  //Calculate maximal value of data
  double max = List[0];
  for(int j=1; j < List.length; j++)
    if (List[j] > max)  max = List[j];
  System.out.println( "\nMax value = " + max);
  //Calculates number of classes
```

```
  int nClasses  = (int) ((max- infLimit)/intervalLength)+1;
  System.out.println( "Number of classes = " + nClasses);
  //Calculates interval borders and class markers
  //Grouping mode 2: defines inferior limit and length
  //of intervals.
  double classMarkers[] = new double[nClasses];
  double  Borders[] = new double[nClasses+2];
  Borders[0] = infLimit;
  classMarkers[0] = infLimit + intervalLength/2;
  for(int j=1; j < nClasses; j++)
    classMarkers[j] = classMarkers[j-1] + intervalLength;
  for(int j=1; j <= nClasses+1; j++)
    Borders[j] = Borders[j-1] + intervalLength;
  System.out.println( "\nBorders are" );
  printVector(Borders,nClasses+1 );
  System.out.println( "\nClass markers are" );
  printVector(classMarkers,nClasses );
 double FreqTable[][] = new double[nClasses][2];
 System.out.println(
 "\nEXTENDED TABLE OF ABSOLUTE FREQUENCIES" );
 //reuse
 FreqTable = group(List, Borders, classMarkers );
 meanVarFreqTable(FreqTable);
}


public static void groupM1()
{
//Declaration of data:
//observed values as a list
//Example 24 pag 12
double    List[] =
{
48.5, 49.2, 51.0, 50.9, 48.7, 50.5, 49.5, 50.8, 50.0, 51.1,
50.7, 51.1, 49.3, 49.1, 50.0, 48.7, 51.2, 49.2, 49.0, 49.3,
49.1, 48.7, 48.6, 49.2, 49.9, 50.1, 50.1, 50.9, 52.4, 50.2,
49.5, 50.9, 51.4, 49.7, 49.8, 50.8, 50.3, 51.8, 50.0, 51.1,
51.3, 50.1, 50.4, 51.1, 49.8, 49.8, 50.1, 50.7, 50.1, 49.9,
49.7, 51.1, 49.7, 49.9, 49.8, 50.5, 49.7, 50.8, 50.7, 50.7,
50.8,  51.5, 51.0
};
System.out.println("\n\n");
System.out.println("FROM DATA LIST TO FREQUENCY TABLE:");
printList(List, "Raw data");
//Borders of the intervals
double  Borders[] = {48.5, 49.5, 50.5, 51.5, 52.5};
```

```
double classMarkers[] = {49, 50, 51, 52};
int nClasses = classMarkers.length;
double FreqTable[][] = new double[nClasses][2];
printList(Borders, "Borders of intervals");
printList(classMarkers, "Class markers");
FreqTable = group(List, Borders, classMarkers );
meanVarFreqTable(FreqTable);
}



public static void groupM2( )
{
 //Declaration of data:
 //observed values as a list
 //Example 24 pag 12
 double    List[] =
{
48.5, 49.2, 51.0, 50.9, 48.7, 50.5, 49.5, 50.8, 50.0, 51.1,
50.7, 51.1, 49.3, 49.1, 50.0, 48.7, 51.2, 49.2, 49.0, 49.3,
49.1, 48.7, 48.6, 49.2, 49.9, 50.1, 50.1, 50.9, 52.4, 50.2,
49.5, 50.9, 51.4, 49.7, 49.8, 50.8, 50.3, 51.8, 50.0, 51.1,
51.3, 50.1, 50.4, 51.1, 49.8, 49.8, 50.1, 50.7, 50.1, 49.9,
49.7, 51.1, 49.7, 49.9, 49.8, 50.5, 49.7, 50.8, 50.7, 50.7,
50.8,  51.5, 51.0
   };
 System.out.println("\n\n");
 System.out.println("FROM DATA LIST TO FREQUENCY TABLE:");
 printList(List, "Raw data");

 //Calculate minimal value of data
 double min = List[0];
 for(int j=1; j < List.length; j++)
    if (List[j] < min)  min = List[j];
 System.out.println( "\nMin value = " + min);


 double infLimit = min;
 //Interval length
 double intervalLength = 1;
 System.out.println("Inferior limit = " + infLimit);
 System.out.println("Interval length = "
  + intervalLength);
 group(List, infLimit, intervalLength);
}
```

```
//Data is grouped and processed
public static void groupData( )
{
 //Method 1: all borders and class markers
 // are given by hand.
 groupM1();
 //Method 1: inferior border of first interval and
 //the length  of intervals are given.
 groupM2();
}
}//End of main class
```

54, pag. 50. The function of encoding a frequency table as a graphic for PSTricks is added to the toolbox.

```
//Program F54 toolBox5
/*
 This is a tool box of methods to calculate
 some statistics of a list of data.

 How to use this program:

 1. Append it to any project and run it.
 2. This program is a tool box because its main method
    can be filled with any allowed method.
 3. Most relevant allowed methods are listed in the main
    but only one is activated, all others are
    muted.
 4. To silence one method, append // as a prefix.
    To activate a method, delete the prefix //.
 5. For a given task, choose the best suited method
    among the diverse variants.
 6. To run a method with your own data,
    modify the inbuilt data according to your needs.
 7. This first version has just one aim:
    To implement fundamental tasks as transparently
    as possible. So, it looks rather like an array of
    independent, non interacting  methods.
 8. To inquire about unknown terms, drag the mouse over
    the term and wait until Eclipse displays the
    corresponding documentation.
 9. To go   from the name of a
    method  to its implementation:
     Pose the cursor at the end of the line
     with the name of the method and press the space bar,
     next press F3: you will be transported to the
```

```
     implementation of the method.
 10. To mark a specific site in the program:
     -Activate line numbering.
     -Pose the cursor over the line you want to mark.
     -Go to the corresponding number at the left margin
      of the Editor Window and right click.
     -Choose the Add Bookmark menu and punch it.
     -A blue marker will appear in the right margin of the
      Editor Window. By clicking over it, one can
      return to the desired place.
      One can also play with the menu
      Window -> ShowView -> Outline
      A list with all methods will appear:
      click over any one and you will
      transported to the corresponding place
      along the text of the program.

*/



public class toolBox5
{

 //The outer application is instantiated
 private static toolBox5 p = new toolBox5();




//============================
//MAIN--MAIN--MAIN--MAIN--MAIN
//============================

 public static void main(String[] args)
  {
   meanVarList();
   meanVarFreqTable();
   meanVarCumTable();
   groupData();
   codePSTricks();
  }
```

```
//=============================================
//               IMPLEMENTATIONS
//=============================================




//=============================================
//      Methods of general use
//=============================================


//Prints a list and its title
private static void printList(double DataVector[],
                              String title)
{
System.out.println( "\n " + title );
int nData = DataVector.length;
boolean go = true;
int nCols = 10;
int nRows = 9;
int counter = 0;
for (int i = 0; (i < nRows) & go; i++)
 {
  for (int j = 0; (j < nCols) & go; j++)
   {
    System.out.print( DataVector[i*10 + j] + "   ");
    if (counter >= nData-1) go = false;
    counter = counter+1;
   }
 System.out.println( );
}
System.out.print("\nNumber of data = " +nData + "\n");
}


//Prints n data in vector
private static void printVector(double Vect[], int n)
{
//System.out.println( "Data are" );
for ( int i = 0; i < n; i++ )
   System.out.println( Vect[ i ]  );
}
```

```
//Prints  a matrix and is title
private static void printMatrix(double Matrix[][],
                        String title)
{
System.out.println( title );
int m = Matrix.length;
 for ( int i = 0; i < m; i++ )
 System.out.println( Matrix[ i ][0]+
        "\t "+ Matrix[ i ][1]);
}


//Documentation is printed
private static void explanation()
{
 System.out.println( "\nCoeff around 0.01, " +
                     "homogeneous population." +
                     "\nCoeff around 0.1, " +
                     " +/- homogeneous population."+
                     "\nCoeff around 0.5, " +
                     "  heterogenous population.");
}


//=============================================
//       MEAN AND VAR O A LIST OF DATA
//=============================================



 //======== Inner class definition=======

 //This inner class defines a new type: Table.
 //It converts array F (two dimensional) into
 //a class or object, ready for reuse.
 private  class Table
 {
  //The class Table fundamentally has three elements:
  //a matrix with two columns and its length,
  //which is the number of rows and
  //the titles of columns.
  int length;
  String[] Titles = { "x",  "x2 = x^2"};
  double M[][] = new double[2000][2];

  //The input of Table is defined here
```

```
  //by means of an initialization procedure
  //that is called constructor.
  //The natural input is the raw data,
  //which is consigned in a one dimensional array.
  Table(double[] V )
  {
   length = V.length;
   //The matrix M of Table is initialized
   //Raw data go in the first column.
   //Squares in the second.
   for(int i = 0; i < length; i++)
   {
 M[i][0] = V[i];
 M[i][1] = V[i]*V[i];
   }
  }

  //Data of column k are summed.
  //The input is a table, the output is of type double.
  private  double sum(Table  T, int k)
  {
   double sum = 0;
   for(int i = 0; i < T.length; i++)
  sum  = sum +  T.M[i][k];
   return sum;
  }

 private  void printTable(Table T)
  {

    System.out.printf("%n %10s  %11s", T.Titles [0], T.Titles [1]);
 for ( int i = 0; i < T.length; i++ )
    System.out.printf("%n %10.3f   %10.3f ",
        T.M[i][0], T.M[i][1] );
 System.out.println( );
  }
}//end of inner class


//=== Methods of the outer class toolBox5 ===


//This is the core of the main class
//Here, commands are given to execute the overall task.
private static void meanVarVector(double[] V)
{
```

```
 //A table is declared
 Table T =  p.new   Table(V);
 T.printTable(T);
 double sumx = T.sum(T,0);
 int n = T.length;
 double mean = sumx/ n;
 double sumx2 = T.sum(T,1);
 System.out.printf("%n Sum %6.3f   %10.3f ",
                sumx,        sumx2 );
 System.out.print("\n \n");
 System.out.println("n = number of data = " + n);

 System.out.println("\nsumx = " + sumx);
 System.out.println("Mean = sumx / n");
 System.out.println("Mean = " + mean);
 System.out.println("sumx2 = " + sumx2);
 System.out.println("variance = " +
   "(n * sumx2 - sumx*sumx)/ (n*( n-1))");
 double var = (n * sumx2 - sumx*sumx)/ (n*( n-1));
 System.out.println( "Variance = "+ var);
 double deviation = Math.pow( var, 0.5);
 System.out.println( "Deviation = "+ deviation);
 double coeffVar =  deviation/mean;
 System.out.println( "Coefficient of variation " +
                "= deviation/mean");
 System.out.println( "Coefficient of variation = "
                                + coeffVar);
 explanation();
}

//A list of data  is  processed
private static void meanVarList( )
{
  System.out.println("\n\n");
  System.out.println( "MEAN AND VAR OF A LIST OF DATA");
  //Example 19 page 9.
  double  List[]  = {5, 3, 4.5, 2};
  //Prints list and title
  printList(List,  "Raw data");
  System.out.println( "\nTABLE WITH x AND ITS SQUARE");
  meanVarVector(List);
}

//================================================
// MEAN AND VAR OF A TABLE OF ABSOLUTE FREQUENCIES
//================================================
```

```
//======== Inner class definition=======

//This inner class defines a new type: Table.
//It converts array F (two dimensional) into
//a class or object, ready for reuse.
private  class TableFreq
 {
//The class Table has two main elements:
//1) The titles of  the columns, whose length
//gives the number of columns.
//2) A matrix, whose length is the number of rows.

String[] Titles = { "x",  "F",   "xF", "x2F"};
double M[][] = new double[2000][Titles.length];
int nRows;
//The input of Table is defined here
//by means of an initialization procedure
//that is called constructor.
//The natural input is the frequency table
//which is consigned in a matrix FT with 2 columns.
TableFreq(double[][] FT )
 {
   nRows = FT.length;
   //The matrix M of Table is initialized
   //Raw data go in   columns 0 and 1.
   //Squares in the second.
   for(int i = 0; i < nRows; i++)
   {
 M[i][0] = FT[i][0];
 M[i][1] = FT[i][1];
 M[i][2] = M[i][0] * M[i][1];
 M[i][3] = M[i][0] * M[i][2];
   }
  }

//Data of column k are summed.
//The input is a table, the output is of type double.
private  double sum(TableFreq  T, int k)
 {
  double sum = 0;
  for(int i = 0; i < T.nRows; i++)
  sum  = sum +  T.M[i][k];
  return sum;
 }
```

```
private  void printTable(TableFreq T)
 {
  System.out.println("\nExtended matrix to " +
                "calculate variance:");
  System.out.printf("%n %10s %10s %10s %10s",
            T.Titles [0], T.Titles [1],
             T.Titles [2],T.Titles [3]);
    for ( int i = 0; i < T.nRows; i++ )
 System.out.printf(
"%n %10.3f %10.3f %10.3f %10.3f ",
T.M[i][0], T.M[i][1], T.M[i][2], T.M[i][3] );
  System.out.println( );
 }
}//end of inner class


//=== Methods of the outer class toolBox5  ===



// Mean and variance of a frequency table
private static void meanVarFreqTable(double FreqTable[][])
{
//A table is declared
TableFreq T =  p.new   TableFreq(FreqTable);
T.printTable(T);
double n = T.sum(T,1);
double sumxF = T.sum(T,2);
double sumx2F = T.sum(T,3);
System.out.printf("Sum %18.3f %10.3f %10.3f ",
             n,   sumxF,        sumx2F );
double mean = sumxF/ n;
System.out.print(
  "\n\n n = Number of  objects = sum F = "+ (int) n);
System.out.printf( "\n sum xF  = %10.3f ", sumxF);
System.out.printf( "\n sum x2F  = %10.3f ",  sumx2F);
System.out.printf( "\n Mean = sum xF / n = %10.3f ", mean);
double Sxx  = sumx2F -    Math.pow(sumxF, 2)/n;
System.out.printf(
 "\n Sxx = sum x2F - (sumxF)^2 /n = %10.3f ", Sxx);
double var = Sxx / (n-1);
System.out.printf(
 "\n Variance =  Sxx/(n-1) = %10.3f ", var);
double deviation = Math.sqrt(var);
System.out.printf(
```

```
 "\n Deviation =  %10.3f ", deviation);
double c = deviation/ mean;
System.out.printf(
   "\n Coefficient of variation =  %10.3f \n", c);
explanation();
}


//A Frequency Table is processed
private static void meanVarFreqTable()
{
System.out.println("\n\n");
System.out.println("\n\nMEAN-VAR OF A FREQUENCY TABLE");
//Frequency table: first coordinate = class marker
//second coordinate = frequency

//Example 20, pag 10
double  FreqT[][] =
        {{4, 3},
         {7, 6},
         {10, 10},
         {13, 15},
         {16, 4},
         {19, 2}};

printMatrix(FreqT, "\n Raw data");
meanVarFreqTable(FreqT);
}


//=================================================
//MEAN AND VAR OF A TABLE OF CUMULATIVE FREQUENCIES
//=================================================


private static void meanVarCumTable(double[][] cumFreqT)
{
 printMatrix(cumFreqT, "\n Raw data");
 //Absolute frequencies are calculated
 //and kept in FreqT, a matrix with two columns.
 double FreqT[][] = new double[cumFreqT.length][2];

 FreqT[0] = cumFreqT[0];
 int lx = cumFreqT.length;
 for(int i = 1; i < lx;  i++)
 {
```

```
  FreqT[i][0] = cumFreqT[i][0];
  FreqT[i][1] = cumFreqT[i][1]-cumFreqT[i-1][1];
 }
 System.out.println("\n\n");
 System.out.println(
 "\n MEAN-VAR OF ASSOCIATED  FREQUENCY TABLE");
  meanVarFreqTable(FreqT);//reuse
 }

//A cumulative frequency table is processed
public static void meanVarCumTable()
{
 System.out.println(
 "MEAN-VAR OF A CUMULATIVE FREQUENCY TABLE");
 //Cumulative table: first coordinate = class marker
 //second coordinate = cumulative frequency

 //Problem 1 of version 1,  pag 42.
 double  cumFreqT[][] =
        {{32,7},
         {40,12},
         {45,17},
         {50,45},
         {55,52},
         {60,58}};
  meanVarCumTable(cumFreqT);
}


//==================================================
// GROUPING OF DATA IN INTERVALS + MEAN & VAR
//==================================================



//List is the input, a frequency table is the output
private static double[][] group(double List[],
        double Borders[], double classMarkers[]  )
{
 int nClasses = classMarkers.length;
 double FreqTable[][] = new double[nClasses][2];
 int nData = List.length;

 for(int j=0; j < nClasses; j++)
  {
FreqTable[j][0] = classMarkers[j];
```

```
FreqTable[j][1] = 0;
  }
 for(int i=0; i< nData; i++)
  {
   for(int j=0; j < nClasses; j++)
 {
  if((Borders[j] <= List[i]) & (List[i] < Borders[j+1]))
  FreqTable[j][1] =FreqTable[j][1] +1;
}

 }

 return FreqTable;
}




//Data is grouped into a frequency table.
//The borders and class marker of each interval
//is calculated.
//Final output = Mean and var of the freq table.
private static void group(double List[],
double infLimit, double intervalLength)
{
  //Calculate minimal value of data
  double min = List[0];
  for(int j=1; j < List.length; j++)
   if (List[j] < min)  min = List[j];
  //Calculate maximal value of data
  double max = List[0];
  for(int j=1; j < List.length; j++)
    if (List[j] > max)  max = List[j];
  System.out.println( "\nMax value = " + max);
  //Calculates number of classes
  int nClasses  = (int) ((max- infLimit)/intervalLength)+1;
  System.out.println( "Number of classes = " + nClasses);
  //Calculates interval borders and class markers
  //Grouping mode 2: defines inferior limit and length
  //of intervals.
  double classMarkers[] = new double[nClasses];
  double  Borders[] = new double[nClasses+2];
  Borders[0] = infLimit;
  classMarkers[0] = infLimit + intervalLength/2;
  for(int j=1; j < nClasses; j++)
```

```
    classMarkers[j] = classMarkers[j-1] + intervalLength;
  for(int j=1; j <= nClasses+1; j++)
    Borders[j] = Borders[j-1] + intervalLength;
  System.out.println( "\nBorders are" );
  printVector(Borders,nClasses+1 );
  System.out.println( "\nClass markers are" );
  printVector(classMarkers,nClasses );
 double FreqTable[][] = new double[nClasses][2];
 System.out.println(
 "\nEXTENDED TABLE OF ABSOLUTE FREQUENCIES" );
 //reuse
 FreqTable = group(List, Borders, classMarkers );
 meanVarFreqTable(FreqTable);
}


public static void groupM1()
{
//Declaration of data:
//observed values as a list
//Example 24 pag 12
double    List[] =
{
48.5, 49.2, 51.0, 50.9, 48.7, 50.5, 49.5, 50.8, 50.0, 51.1,
50.7, 51.1, 49.3, 49.1, 50.0, 48.7, 51.2, 49.2, 49.0, 49.3,
49.1, 48.7, 48.6, 49.2, 49.9, 50.1, 50.1, 50.9, 52.4, 50.2,
49.5, 50.9, 51.4, 49.7, 49.8, 50.8, 50.3, 51.8, 50.0, 51.1,
51.3, 50.1, 50.4, 51.1, 49.8, 49.8, 50.1, 50.7, 50.1, 49.9,
49.7, 51.1, 49.7, 49.9, 49.8, 50.5, 49.7, 50.8, 50.7, 50.7,
50.8,  51.5, 51.0
};
System.out.println("\n\n");
System.out.println("FROM DATA LIST TO FREQUENCY TABLE:");
printList(List, "Raw data");
//Borders of the intervals
double  Borders[] = {48.5, 49.5, 50.5, 51.5, 52.5};
double classMarkers[] = {49, 50, 51, 52};
int nClasses = classMarkers.length;
double FreqTable[][] = new double[nClasses][2];
printList(Borders, "Borders of intervals");
printList(classMarkers, "Class markers");
FreqTable = group(List, Borders, classMarkers );
meanVarFreqTable(FreqTable);
}
```

```
public static void groupM2( )
{
 //Declaration of data:
 //observed values as a list
 //Example 24 pag 12
 double    List[] =
{
48.5, 49.2, 51.0, 50.9, 48.7, 50.5, 49.5, 50.8, 50.0, 51.1,
50.7, 51.1, 49.3, 49.1, 50.0, 48.7, 51.2, 49.2, 49.0, 49.3,
49.1, 48.7, 48.6, 49.2, 49.9, 50.1, 50.1, 50.9, 52.4, 50.2,
49.5, 50.9, 51.4, 49.7, 49.8, 50.8, 50.3, 51.8, 50.0, 51.1,
51.3, 50.1, 50.4, 51.1, 49.8, 49.8, 50.1, 50.7, 50.1, 49.9,
49.7, 51.1, 49.7, 49.9, 49.8, 50.5, 49.7, 50.8, 50.7, 50.7,
50.8,  51.5, 51.0
   };
 System.out.println("\n\n");
 System.out.println("FROM DATA LIST TO FREQUENCY TABLE:");
 printList(List, "Raw data");

 //Calculate minimal value of data
 double min = List[0];
 for(int j=1; j < List.length; j++)
    if (List[j] < min)  min = List[j];
 System.out.println( "\nMin value = " + min);


 double infLimit = min;
 //Interval length
 double intervalLength = 1;
 System.out.println("Inferior limit = " + infLimit);
 System.out.println("Interval length = "
  + intervalLength);
 group(List, infLimit, intervalLength);
}

//Data is grouped and processed
public static void groupData( )
{
 //Method 1: all borders and class markers
 // are given by hand.
 groupM1();
 //Method 1: inferior border of first interval and
 //the length  of intervals are given.
 groupM2();
}
```

```
//*****************************
//Pstricks code
//*****************************


//Produces the pstricks  code for the bar graph
//of a frequency table
public static void makeCode(double FreqTable[][])
{
 System.out.println("\nPSTRICKS code for the FreqTable\n");
 int n = FreqTable.length;
 //Width of each column
 double deltaX = 4;
 //The minimum x-coordinate is found
 double Minxx = 1000;
 for ( int i = 0; i < n; i++ )
  if (FreqTable[i][0] < Minxx)
        Minxx =   FreqTable[i][0];
 int Minxxx = (int) (deltaX*Minxx - deltaX);
 //The maximum x-coordinate is found
 double Maxx = 0;
 for ( int i = 0; i < n; i++ )
  if ( FreqTable[i][0] > Maxx)
      Maxx =    FreqTable[i][0];
 int Maxxx = (int) (deltaX*Maxx + deltaX);
 //The maximum y-coordinate is found
 int Maxyy = 0;
 for ( int i = 0; i < n; i++ )
  if (FreqTable[i][1] > Maxyy)
       Maxyy = (int) FreqTable[i][1] + 2;
 //Starting clause
 //x- and y-units.
 //The needed space is defined
 System.out.println("\\begin{center}" +
     "\n\\psset{xunit=0.15,yunit=0.3}" +
 "\n\\begin{pspicture}(" + Minxxx +",0)(" +
 Maxxx+ "," + Maxyy+")");
 //A rectangle is drawn for each entry in the
 //frequency table

 int lim1,lim2;
 for ( int i = 0; i < n; i++ )
```

```
  {
   lim1 = (int) (deltaX*FreqTable[i][0] -deltaX/2);
   lim2 = (int) (lim1 + deltaX);
   System.out.print(
 "\n\\psline("+ lim1 + ",0)" +
     "("+lim1 +"," + FreqTable[i][1]+ ")" +
     "("+lim2 + ","+ FreqTable[i][1]+ ")" +
     "("+lim2 + ","+ "0)"
    );
  }

 //Horizontal axis
 System.out.println("\n\\psline(" +
      Minxxx +",0)("+ Maxxx + ",0)");

 //Class markers (first coordinates) are indicated
 for ( int i = 0; i < n; i++ )
   {
int xposition = (int) (deltaX* FreqTable[i][0]);
int yposition = - 1;
    System.out.print
 (
  "\n\\rput*("+ xposition + "," + yposition+ "){"+
   (int) (FreqTable[i][0])+ "}"
 );
   }
 System.out.println();

 //Labels for heights (second coordinates) are indicated
 for ( int i = 0; i < n; i++ )
  {
   int xposition = (int) (deltaX* FreqTable[i][0]);
   int yposition = (int) (1+ FreqTable[i][1]);
   System.out.print
(
  "\n\\rput*("+ xposition + "," + yposition+ "){" +
      (int) (FreqTable[i][1])+ "}"
 );
 }

 //End clause
 System.out.println( "\n\\end{pspicture}");
 System.out.println( "\\end{center}");
 }

public static void codePSTricks()
```

```
 {
  double  FreqT[][] =
        {{3, 3},
         {4, 3},
         {5, 2},
         {6, 2},
         {7, 1},
         {8, 1}};
 printMatrix(FreqT, "\n\nFREQUENCY  TABLE");
 makeCode(FreqT);
 }
}//End of main class
```

Problems of Chapter 3

59, pag. 56. The binomial distribution following a gossipless, transparent, functional style:

```
//Program F59 binomial2
//This program outputs p(x)
//for given n and p,
//Style = gossipless, transparent, functional.

public class binomial2
{

//Prints n data in vector
private static void printVector(double[] Vect, int n)
{
 //System.out.println( "Data are" );
 for ( int i = 0; i < n; i++ )
   System.out.println( Vect[ i ]  );
}

//Sums data in vector with real numbers
private static double sumVector(double[] Vect, int n)
{
 double sum = 0;
 for ( int i = 0; i < n; i++ )
sum = sum + Vect[i];
 return sum;
```

```
}

//Returns numb!
private static long factorial (long numb)
{
 if (numb <=1 ) return 1;
 else
 return numb * factorial (numb -1);
}


//The  binomial distribution
//for  n repeats
//probability of head p,
//amplification factor F.
private static double[]
     binomialDistribution(int n, double p, double F  )
{
 double[] pBinomial = new double[n+1];
 double pBinomialF[] = new double[n+1];
 double CoefficientsBinomial[] = new double[1000];
 long biCoeff = 0;
 for(int r = 0; r < n+1; r++)
  {
   biCoeff = factorial(n) /
       (factorial(r) * factorial(n-r));
   CoefficientsBinomial[r] = biCoeff;
  }
 System.out.println("\nCoefficients ");
 printVector(CoefficientsBinomial,n+1);
 System.out.println("Sum = " +
     sumVector(CoefficientsBinomial,n+1));

 for(int r = 0; r < n+1; r++)
  {
   pBinomial[r] = CoefficientsBinomial[r]  *
        Math.pow(p, r)* Math.pow(1-p, n-r);
  }
 System.out.println("\nProbabilities for 0,1,2...");
```

```
 System.out.println("Paste this vector to Excel" +
           " and make a graphic");
 printVector(pBinomial,n+1);
 sumVector(pBinomial,n+1);
 System.out.println("Sum = "
            +  sumVector(pBinomial,n+1));
 for(int k = 0; k < n+1; k++)
 pBinomialF[k] =  F* pBinomial[k];
 System.out.println("\nAbsolute frequencies for F = " + F);
 printVector(pBinomialF,n+1);
 System.out.println("Sum = "  +
           sumVector(pBinomialF,n+1));
 return pBinomial;
}


//Detailed computation of the mean
private static void detailedMean(int n,
      double p,
      double[] pBinomial)
{
 System.out.println("\nDetailed mean: xp(x)");
 double  sum =0;
 double term;
 for(int r = 0; r < n+1; r++)
 {
  term =  r *  pBinomial[r];
  System.out.println( r + "    " + term) ;
  sum = sum +term;
 }
 System.out.println( "mean = sum xp(x) = " + sum) ;
 System.out.println( "Expected mean = np =  " + n*p) ;
}

//Detailed computation of the variance
private static void detailedVariance(int n,
      double p,
      double[] pBinomial)
{
```

```
 System.out.println("\nDetailed variance: " +
  "(x-mean)*(-mean)*p(x)");
 double  sum =0;
 double term;
 for(int r = 0; r < n+1; r++)
  {
   term =  Math.pow(r- n*p,2) *  pBinomial[r];
   System.out.println( r + "    " +  term) ;
   sum = sum +term;
  }
 System.out.println( "Variance = " +
"sum (x-mean)*(-mean)*p(x) = " + sum) ;
 System.out.println(
"Expected var = np(1-p) =  " + n*p*(1-p)) ;
 System.out.println( "Deviation = " + Math.sqrt(sum));
}

private static void binomialDist()
{
 System.out.println("BINOMIAL DISTRIBUTION ");
 double pBinomial[] = new double[1000];
 int n= 7;
 double p = 0.3;
 int F = 35;
 System.out.println("n= " + n);
 System.out.println("p = " + p);
 System.out.println("F = " + F);
 pBinomial = binomialDistribution(n,p, F);
 detailedMean(n,p,pBinomial);
 detailedVariance(n,p,pBinomial);
}

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

}
```

60, pag. 56. The code to calculate a binomial distribution together with mean, variance and probability of a closed interval.

```
//Program F60 binomial3
//This program outputs p(x)
//for given n and p.
//Mean, variance and probability of a closed interval
//are also enabled.
//Style = gossipless, transparent, functional.


public class binomial3
{

//Prints n data in vector
private static void printVector(double[] Vect, int n)
{
 //System.out.println( "Data are" );
 for ( int i = 0; i < n; i++ )
   System.out.println( Vect[ i ]  );
}

//Sums data in vector with real numbers
private static double sumVector(double[] Vect, int n)
{
 double sum = 0;
 for ( int i = 0; i < n; i++ )
sum = sum + Vect[i];
 return sum;
}

//Returns numb!
private static long factorial (long numb)
{
 if (numb <=1 ) return 1;
 else
 return numb * factorial (numb -1);
}
```

```
//The  binomial distribution
//for  n repeats
//probability of head p,
//amplification factor F.
private static double[]  binomialDistribution(int n,
          double p, double F  )
{
 double[] pBinomial = new double[n+1];
 double[] pBinomialF = new double[n+1];
 double[] CoefficientsBinomial = new double[1000];
 long biCoeff = 0;
 for(int r = 0; r < n+1; r++)
  {
   biCoeff = factorial(n) /
       (factorial(r) * factorial(n-r));
   CoefficientsBinomial[r] = biCoeff;
  }
 System.out.println("\nCoefficients ");
 printVector(CoefficientsBinomial,n+1);
 System.out.println("Sum = " +
     sumVector(CoefficientsBinomial,n+1));

 for(int r = 0; r < n+1; r++)
  {
   pBinomial[r] = CoefficientsBinomial[r]  *
        Math.pow(p, r)* Math.pow(1-p, n-r);
  }
 System.out.println("\nProbabilities for 0,1,2...");
 System.out.println("Paste this vector to Excel" +
          " and make a graphic");
 printVector(pBinomial,n+1);
 sumVector(pBinomial,n+1);
 System.out.println("Sum = "
           +  sumVector(pBinomial,n+1));
 for(int k = 0; k < n+1; k++)
 pBinomialF[k] =  F* pBinomial[k];
 System.out.println("\nAbsolute frequencies for F = " + F);
 printVector(pBinomialF,n+1);
```

```
 System.out.println("Sum = "  +
           sumVector(pBinomialF,n+1));
 return pBinomial;
}

//Probability of closed region (extremes included)
private static double pClosedRegion(int n,
      double p, int limitInf, int limitMax,
      double[] pBinomial)
{
 double  sum =0;
 for(int r = limitInf; r <= limitMax; r++)
  sum =  sum +  pBinomial[r];

 return sum;
}

//Detailed computation of the mean
private static void detailedMean(int n,
      double p,
      double[] pBinomial)
{
 System.out.println("\nDetailed mean: xp(x)");
 double  sum =0;
 double term;
 for(int r = 0; r < n+1; r++)
 {
  term =  r *  pBinomial[r];
  System.out.println( r + "    " + term) ;
  sum = sum +term;
 }
 System.out.println( "mean = sum xp(x) = " + sum) ;
 System.out.println( "Expected mean = np =  " + n*p) ;
}

//Detailed computation of the variance
private static void detailedVariance(int n,
      double p,
      double[] pBinomial)
```

```
{
 System.out.println("\nDetailed variance: " +
  "(x-mean)*(-mean)*p(x)");
 double  sum =0;
 double term;
 for(int r = 0; r < n+1; r++)
  {
   term =  Math.pow(r- n*p,2) *  pBinomial[r];
   System.out.println( r + "    " +  term) ;
   sum = sum +term;
  }
 System.out.println( "Variance = " +
"sum (x-mean)*(-mean)*p(x) = " + sum) ;
 System.out.println(
"Expected var = np(1-p) =  " + n*p*(1-p)) ;
 System.out.println( "Deviation = " + Math.sqrt(sum));
}

private static void binomialDist()
{
 System.out.println("BINOMIAL DISTRIBUTION ");
 double pBinomial[] = new double[1000];
 int n= 5;
 double p = 0.2;

 int F = 35;
 System.out.println("n= " + n);
 System.out.println("p = " + p);
 System.out.println("F = " + F);
 pBinomial = binomialDistribution(n,p, F);
 detailedMean(n,p,pBinomial);
 detailedVariance(n,p,pBinomial);
 //Probability of a closed region (extremes included)
 int limitInf = 2;
 int limitMax = 4;
 double pR = pClosedRegion(n,p,limitInf,limitMax,pBinomial);
 System.out.println( "\nProb of closed region in " +
  "within "+ limitInf + "  and " + limitMax +
     ",\nextremes included = " + pR) ;
```

```
}

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

}
```

62, pag. 57.

```
//Program F62 binomial4
//This program outputs p(x)
//for given n and p.
//Mean, variance and probability of a closed interval
//and a solution to the inverse problem
//are also enabled.
//Style =  gossipless, transparent, functional.


public class binomial4
{

//Prints n data in vector
private static void printVector(double[] Vect, int n)
{
 //System.out.println( "Data are" );
 for ( int i = 0; i < n; i++ )
   System.out.println(i + "\t" + Vect[ i ]  );
}

//Sums data in vector with real numbers
private static double sumVector(double[] Vect, int n)
{
 double sum = 0;
 for ( int i = 0; i < n; i++ )
sum = sum + Vect[i];
```

```
 return sum;
}


//The  binomial distribution
//for  n repeats
//probability of head p,
//amplification factor F.
//Arithmetic trickery in action.
private static double[]  binomialDistribution(int n,
          double p, double F  )
{
double[] pBinomial = new double[n+1];
double[] pBinomialF = new double[n+1];
double[] CoefficientsBinomial = new double[n+1];
double biCoeff = 1;
CoefficientsBinomial[0] = biCoeff;
for(int r = 1; r < n+1; r++)
{
 biCoeff = biCoeff * (n+1-r) / r;
 CoefficientsBinomial[r] = biCoeff;
}
System.out.println("\nCoefficients ");
printVector(CoefficientsBinomial,n+1);
System.out.println("Sum = " +
     sumVector(CoefficientsBinomial,n+1));

for(int r = 0; r < n+1; r++)
{
 pBinomial[r] = CoefficientsBinomial[r]  *
     Math.exp( r* Math.log(p) + (n-r) * Math.log(1-p));
}
System.out.println("\nProbabilities for 0,1,2...");
System.out.println("Paste this vector to Excel" +
           " and make a graphic");
printVector(pBinomial,n+1);
sumVector(pBinomial,n+1);
System.out.println("Sum = "
            +  sumVector(pBinomial,n+1));
```

```
for(int k = 0; k < n+1; k++)
 pBinomialF[k] =  F* pBinomial[k];
System.out.println("\nAbsolute frequencies for F = " + F);
printVector(pBinomialF,n+1);
System.out.println("Sum = "  +
          sumVector(pBinomialF,n+1));
return pBinomial;
}



//Probability of closed region (extremes included)
private static double pClosedRegion( int limitInf,
             int limitMax,  double[] pBinomial)
{
 double  sum =0;
 for(int r = limitInf; r <= limitMax; r++)
  sum =  sum +  pBinomial[r];

 return sum;
}

//Detailed computation of the mean
private static void detailedMean(int n,
      double p,
      double[] pBinomial)
{
 System.out.println("\nDetailed mean: xp(x)");
 double  sum =0;
 double term;
 for(int r = 0; r < n+1; r++)
 {
  term =  r *  pBinomial[r];
  System.out.println( r + "    " + term) ;
  sum = sum +term;
 }
 System.out.println( "mean = sum xp(x) = " + sum) ;
 System.out.println( "Expected mean = np =  " + n*p) ;
}
```

```
//Detailed computation of the variance
private static void detailedVariance(int n,
      double p,
      double[] pBinomial)
{
 System.out.println("\nDetailed variance: " +
  "(x-mean)*(-mean)*p(x)");
 double  sum =0;
 double term;
 for(int r = 0; r < n+1; r++)
  {
   term =  Math.pow(r- n*p,2) *  pBinomial[r];
   System.out.println( r + "    " +  term) ;
   sum = sum +term;
  }
 System.out.println( "Variance = " +
"sum (x-mean)*(-mean)*p(x) = " + sum) ;
 System.out.println(
"Expected var = np(1-p) =  " + n*p*(1-p)) ;
 System.out.println( "Deviation = " + Math.sqrt(sum));
}


//Critical value for significance level alpha, one tail
private static void  critical1(  double[] pBinomial,
                                 double alpha )
{
 int numbTails = 1;
 System.out.println("\nLower tail");
 double cumulative = 0;
 double target = alpha;
 int i=-1;
 while( cumulative < target)
  {
   i = i+1;
   cumulative = cumulative + pBinomial[i];
   System.out.println(i + "  cumulative = " + cumulative);
  }
```

```
 System.out.println("To cumulate in the lower tail "
    + target
+ ", \nwe need the first " + i  + " Terms.");
 System.out.println("The critical value for alpha =  "
   + alpha
   + ", numb of tails = " + numbTails
   + "\nin the lower tail is "
   + i+ ". All terms below it are extreme.\n");
 System.out.println("Upper tail");
 int n = pBinomial.length;
 i= n;
 cumulative = 0;
 while( cumulative < target)
  {
   i = i-1;
   cumulative = cumulative + pBinomial[i];
   System.out.println(i + "  cumulative = " + cumulative);
  }

 System.out.println("To cumulate in the upper tail "
    + target
+ ", \nwe need term  " + i  + " and above.");
 System.out.println("The critical value for alpha =  "
   + alpha
   + ", numb of tails = " + numbTails
   + "\nin the upper tail is "
   + i + ". All terms above it are extreme.\n");
}


//Critical value for significance level alpha, two tails
private static  void  critical2(double[] pBinomial,
                                double alpha)
{
 critical1(pBinomial,alpha/2);
}
```

```
//Critical value for significance level alpha, one tail
private static void  critical1( int n, double p,
                                double alpha )
{
int numbTails = 1;
System.out.println("\nLower tail");
double cumulative = 0;
double target = alpha;
int i=-1;
double pBinomial[] = new double[n+1];
pBinomial = binomialDistribution(n,p, 1);
while( cumulative < target)
{
 i = i+1;
 cumulative = cumulative + pBinomial[i];
 System.out.println(i + "  cumulative = " + cumulative);
}
System.out.println("To cumulate in the lower tail "
    + target
+ ", \nwe need the first " + i  + " Terms.");
System.out.println("The critical value for alpha =  "
   + alpha
   + ", numb of tails = " + numbTails
   + "\nin the lower tail is "
   + i+ ". All terms below it are extreme.\n");
System.out.println("Upper tail");
i= n+1;
cumulative = 0;
while( cumulative < target)
{
 i = i-1;
 cumulative = cumulative + pBinomial[i];
 System.out.println(i + "  cumulative = " + cumulative);
}

System.out.println("To cumulate in the upper tail "
    + target
+ ", \nwe need term  " + i  + " and above.");
System.out.println("The critical value for alpha =  "
```

```
   + alpha
   + ", numb of tails = " + numbTails
   + "\nin the upper tail is "
   + i + ". All terms above it are extreme.\n");
}


//Critical value for significance level alpha, two tails
private static  void  critical2(int n, double p,
                                double alpha)
{
critical1(n,p,alpha/2);
}


private static void binomialDist()
{
 System.out.println("BINOMIAL DISTRIBUTION ");

 int n= 1000;
 double p = 0.2;
 double[] pBinomial = new double[n+1];
 int F = 35;
 System.out.println("n= " + n);
 System.out.println("p = " + p);
 System.out.println("F = " + F);
 pBinomial = binomialDistribution(n,p, F);
 detailedMean(n,p,pBinomial);
 detailedVariance(n,p,pBinomial);
 //Probability of a closed region (extremes included)
 int limitInf = 2;
 int limitMax = 4;
 double pR = pClosedRegion(limitInf,limitMax,pBinomial);
 System.out.println( "\nProb of closed region in " +
  "within "+ limitInf + "  and " + limitMax +
     ",\nextremes included = " + pR) ;
 //Inverse problem: find critical values
 double alpha = 0.02;
 System.out.println("Binomial distribution: \n" +
```

```
" n = " + n + ", p = " + p + ", " +
"significance level = alpha = " + alpha);
 System.out.println("\nCRITICAL VALUES");
 System.out.println("\nONE TAIL, alpha = " +alpha);
 System.out.println("\nTerms and cumulative values");
 critical1(pBinomial,alpha);
 critical1(n,p, alpha);
 System.out.println("\nTWO TAILS, alpha = " +alpha);
 System.out.println("\nTerms and cumulative values");
 critical2(pBinomial, alpha);
 critical2(n,p, alpha);
}


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

}
```

63, pag. 57. This is toolBox6:

```
//Program F63 toolBox6
/*
 This is a tool box of methods to calculate
  a) Some statistics of a list of data.
  b) The Pstricks code for a bar diagram
  c) The binomial distribution and its critical values

 How to use this program:

 1. Append it to any project and run it.
 2. This program is a tool box because its main method
    can be filled with any allowed method.
 3. Most relevant allowed methods are listed in the main
    but only one is activated, all others are
    muted.
```

```
 4. To silence one method, append // as a prefix.
    To activate a method, delete the prefix //.
 5. For a given task, choose the best suited method
    among the diverse variants.
 6. To run a method with your own data,
    modify the inbuilt data according to your needs.
 7. This first version has just one aim:
    To implement fundamental tasks as transparently
    as possible. So, it looks rather like an array of
    independent, non interacting  methods.
 8. To inquire about unknown terms, drag the mouse over
    the term and wait until Eclipse displays the
    corresponding documentation.
 9. To go   from the name of a
    method  to its implementation:
     Pose the cursor at the end of the line
     with the name of the method and press the space bar,
     next press F3: you will be transported to the
     implementation of the method.
10. To mark a specific site in the program:
    -Activate line numbering.
    -Pose the cursor over the line you want to mark.
    -Go to the corresponding number at the left margin
     of the Editor Window and right click.
    -Choose the Add Bookmark menu and punch it.
    -A blue marker will appear in the right margin of the
     Editor Window. By clicking over it, one can
     return to the desired place.
     One can also play with the menu
     Window -> ShowView -> Outline
     A list with all methods will appear:
     click over any one and you will
     transported to the corresponding place
     along the text of the program.

*/
```

```
public class toolBox6
{

 //The outer application is instantiated
 private static toolBox6 p = new toolBox6();



//============================
//MAIN--MAIN--MAIN--MAIN--MAIN
//============================

 public static void main(String[] args)
  {
   meanVarList();
   meanVarFreqTable();
   meanVarCumTable();
   groupData();
   codePSTricks();
   binomialDistribution();
  }



//============================================
//               IMPLEMENTATIONS
//============================================



//============================================
//      Methods of general use
//============================================
```

```
//Prints a list and its title
private static void printList(double[] DataVector,
                              String title)
{
System.out.println( "\n " + title );
int nData = DataVector.length;
boolean go = true;
int nCols = 10;
int nRows = 9;
int counter = 0;
for (int i = 0; (i < nRows) & go; i++)
 {
  for (int j = 0; (j < nCols) & go; j++)
   {
    System.out.print( DataVector[i*10 + j] + "   ");
    if (counter >= nData-1) go = false;
    counter = counter+1;
   }
 System.out.println( );
}
System.out.print("\nNumber of data = " +nData + "\n");
}


//Prints n data in vector
private static void printVector(double[] Vect, int n)
{
//System.out.println( "Data are" );
for ( int i = 0; i < n; i++ )
   System.out.println(i + "\t" +  Vect[ i ]  );
}


//Sums data in vector with real numbers
private static double sumVector(double[] Vect, int n)
{
double sum = 0;
for ( int i = 0; i < n; i++ )
sum = sum + Vect[i];
```

```
return sum;
}

//Prints  a matrix and is title
private static void printMatrix(double[][] Matrix,
                        String title)
{
System.out.println( title );
int m = Matrix.length;
 for ( int i = 0; i < m; i++ )
 System.out.println( Matrix[ i ][0]+
        "\t "+ Matrix[ i ][1]);
}


//Documentation is printed
private static void explanation()
{
 System.out.println( "\nCoeff around 0.01, " +
                     "homogeneous population." +
                     "\nCoeff around 0.1, " +
                     " +/- homogeneous population."+
                     "\nCoeff around 0.5, " +
                     "  heterogenous population.");
}


//================================================
//       MEAN AND VAR O A LIST OF DATA
//================================================



 //======== Inner class definition=======

 //This inner class defines a new type: Table.
 //It converts array F (two dimensional) into
 //a class or object, ready for reuse.
 private  class Table
```

```
{
 //The class Table fundamentally has three elements:
 //a matrix with two columns and its length,
 //which is the number of rows and
 //the titles of columns.
 int length;
 String[] Titles = { "x",  "x2 = x^2"};
 double[][] M = new double[2000][2];

 //The input of Table is defined here
 //by means of an initialization procedure
 //that is called constructor.
 //The natural input is the raw data,
 //which is consigned in a one dimensional array.
 Table(double[] V )
 {
  length = V.length;
  //The matrix M of Table is initialized
  //Raw data go in the first column.
  //Squares in the second.
  for(int i = 0; i < length; i++)
  {
M[i][0] = V[i];
M[i][1] = V[i]*V[i];
  }
 }

 //Data of column k are summed.
 //The input is a table, the output is of type double.
 private  double sum(Table  T, int k)
 {
  double sum = 0;
  for(int i = 0; i < T.length; i++)
 sum  = sum +  T.M[i][k];
  return sum;
 }

private  void printTable(Table T)
 {
```

```
    System.out.printf("%n %10s  %11s", T.Titles [0], T.Titles [1]);
 for ( int i = 0; i < T.length; i++ )
    System.out.printf("%n %10.3f   %10.3f ",
        T.M[i][0], T.M[i][1] );
 System.out.println( );
  }
}//end of inner class


//=== Methods of the outer class ===


//This is the core of the main class
//Here, commands are given to execute the overall task.
private static void meanVarVector(double[] V)
{
 //A table is declared
 Table T =  p.new   Table(V);
 T.printTable(T);
 double sumx = T.sum(T,0);
 int n = T.length;
 double mean = sumx/ n;
 double sumx2 = T.sum(T,1);
 System.out.printf("%n Sum %6.3f   %10.3f ",
                sumx,        sumx2 );
 System.out.print("\n \n");
 System.out.println("n = number of data = " + n);

 System.out.println("\nsumx = " + sumx);
 System.out.println("Mean = sumx / n");
 System.out.println("Mean = " + mean);
 System.out.println("sumx2 = " + sumx2);
 System.out.println("variance = " +
   "(n * sumx2 - sumx*sumx)/ (n*( n-1))");
 double var = (n * sumx2 - sumx*sumx)/ (n*( n-1));
 System.out.println( "Variance = "+ var);
 double deviation = Math.pow( var, 0.5);
 System.out.println( "Deviation = "+ deviation);
```

```
 double coeffVar =  deviation/mean;
 System.out.println( "Coefficient of variation " +
                 "= deviation/mean");
 System.out.println( "Coefficient of variation = "
                                  + coeffVar);
 explanation();
}

//A list of data  is  processed
private static void meanVarList( )
{
  System.out.println("\n\n");
  System.out.println( "MEAN AND VAR OF A LIST OF DATA");
  //Example 19 page 9.
  double[]  List  = {5, 3, 4.5, 2};
  //Prints list and title
  printList(List,  "Raw data");
  System.out.println( "\nTABLE WITH x AND ITS SQUARE");
  meanVarVector(List);
}

//=================================================
// MEAN AND VAR OF A TABLE OF ABSOLUTE FREQUENCIES
//=================================================


//======== Inner class definition=======

//This inner class defines a new type: Table.
//It converts array F (two dimensional) into
//a class or object, ready for reuse.
private  class TableFreq
 {
//The class Table has two main elements:
//1) The titles of  the columns, whose length
//gives the number of columns.
//2) A matrix, whose length is the number of rows.

String[] Titles = { "x",  "F",   "xF", "x2F"};
```

```
double[][] M = new double[2000][Titles.length];
int nRows;
//The input of Table is defined here
//by means of an initialization procedure
//that is called constructor.
//The natural input is the frequency table
//which is consigned in a matrix FT with 2 columns.
TableFreq(double[][] FT )
 {
   nRows = FT.length;
   //The matrix M of Table is initialized
   //Raw data go in   columns 0 and 1.
   //Squares in the second.
   for(int i = 0; i < nRows; i++)
   {
 M[i][0] = FT[i][0];
 M[i][1] = FT[i][1];
 M[i][2] = M[i][0] * M[i][1];
 M[i][3] = M[i][0] * M[i][2];
   }
  }

//Data of column k are summed.
//The input is a table, the output is of type double.
private  double sum(TableFreq  T, int k)
 {
  double sum = 0;
  for(int i = 0; i < T.nRows; i++)
  sum  = sum +  T.M[i][k];
  return sum;
 }

private  void printTable(TableFreq T)
 {
  System.out.println("\nExtended matrix to " +
                "calculate variance:");
  System.out.printf("%n %10s %10s %10s %10s",
            T.Titles [0], T.Titles [1],
             T.Titles [2],T.Titles [3]);
```

```
    for ( int i = 0; i < T.nRows; i++ )
 System.out.printf(
"%n %10.3f %10.3f %10.3f %10.3f ",
T.M[i][0], T.M[i][1], T.M[i][2], T.M[i][3] );
  System.out.println( );
 }
}//end of inner class


//=== Methods of the outer class ===



// Mean and variance of a frequency table
private static void meanVarFreqTable(double[][] FreqTable)
{
//A table is declared
TableFreq T =  p.new   TableFreq(FreqTable);
T.printTable(T);
double n = T.sum(T,1);
double sumxF = T.sum(T,2);
double sumx2F = T.sum(T,3);
System.out.printf("Sum %18.3f %10.3f %10.3f ",
             n,   sumxF,        sumx2F );
double mean = sumxF/ n;
System.out.print(
  "\n\n n = Number of  objects = sum F = "+ (int) n);
System.out.printf( "\n sum xF  = %10.3f ", sumxF);
System.out.printf( "\n sum x2F  = %10.3f ",  sumx2F);
System.out.printf( "\n Mean = sum xF / n = %10.3f ", mean);
double Sxx  = sumx2F -    Math.pow(sumxF, 2)/n;
System.out.printf(
 "\n Sxx = sum x2F - (sumxF)^2 /n = %10.3f ", Sxx);
double var = Sxx / (n-1);
System.out.printf(
 "\n Variance =  Sxx/(n-1) = %10.3f ", var);
double deviation = Math.sqrt(var);
System.out.printf(
 "\n Deviation =  %10.3f ", deviation);
```

```
double c = deviation/ mean;
System.out.printf(
   "\n Coefficient of variation =  %10.3f \n", c);
explanation();
}


//A Frequency Table is processed
private static void meanVarFreqTable()
{
System.out.println("\n\n");
System.out.println("\n\nMEAN-VAR OF A FREQUENCY TABLE");
//Frequency table: first coordinate = class marker
//second coordinate = frequency

//Example 20, pag 10
double[][]  FreqT =
        {{4, 3},
         {7, 6},
         {10, 10},
         {13, 15},
         {16, 4},
         {19, 2}};

printMatrix(FreqT, "\n Raw data");
meanVarFreqTable(FreqT);
}


//=================================================
//MEAN AND VAR OF A TABLE OF CUMULATIVE FREQUENCIES
//=================================================


private static void meanVarCumTable(double[][] cumFreqT)
{
 printMatrix(cumFreqT, "\n Raw data");
 //Absolute frequencies are calculated
 //and kept in FreqT, a matrix with two columns.
```

```
 double[][] FreqT = new double[cumFreqT.length][2];

 FreqT[0] = cumFreqT[0];
 int lx = cumFreqT.length;
 for(int i = 1; i < lx;  i++)
 {
  FreqT[i][0] = cumFreqT[i][0];
  FreqT[i][1] = cumFreqT[i][1]-cumFreqT[i-1][1];
 }
 System.out.println("\n\n");
 System.out.println(
 "\n MEAN-VAR OF ASSOCIATED  FREQUENCY TABLE");
  meanVarFreqTable(FreqT);//reuse
 }

//A cumulative frequency table is processed
public static void meanVarCumTable()
{
 System.out.println(
 "MEAN-VAR OF A CUMULATIVE FREQUENCY TABLE");
 //Cumulative table: first coordinate = class marker
 //second coordinate = cumulative frequency

 //Problem 1 of version 1,  pag 42.
 double  cumFreqT[][] =
        {{32,7},
         {40,12},
         {45,17},
         {50,45},
         {55,52},
         {60,58}};
  meanVarCumTable(cumFreqT);
}


//==================================================
// GROUPING OF DATA IN INTERVALS + MEAN & VAR
//==================================================
```

```
//List is the input, a frequency table is the output
private static double[][] group(double[] List,
        double[] Borders, double[] classMarkers  )
{
 int nClasses = classMarkers.length;
 double[][] FreqTable = new double[nClasses][2];
 int nData = List.length;

 for(int j=0; j < nClasses; j++)
  {
FreqTable[j][0] = classMarkers[j];
FreqTable[j][1] = 0;
  }
 for(int i=0; i< nData; i++)
  {
   for(int j=0; j < nClasses; j++)
 {
  if((Borders[j] <= List[i]) & (List[i] < Borders[j+1]))
  FreqTable[j][1] =FreqTable[j][1] +1;
}

 }

 return FreqTable;
}
```

```
//Data is grouped into a frequency table.
//The borders and class marker of each interval
//is calculated.
//Final output = Mean and var of the freq table.
private static void group(double[] List,
double infLimit, double intervalLength)
{
```

```
  //Calculate minimal value of data
  double min = List[0];
  for(int j=1; j < List.length; j++)
   if (List[j] < min)  min = List[j];
  //Calculate maximal value of data
  double max = List[0];
  for(int j=1; j < List.length; j++)
    if (List[j] > max)  max = List[j];
  System.out.println( "\nMax value = " + max);
  //Calculates number of classes
  int nClasses  = (int) ((max- infLimit)/intervalLength)+1;
  System.out.println( "Number of classes = " + nClasses);
  //Calculates interval borders and class markers
  //Grouping mode 2: defines inferior limit and length
  //of intervals.
  double[] classMarkers = new double[nClasses];
  double[]  Borders = new double[nClasses+2];
  Borders[0] = infLimit;
  classMarkers[0] = infLimit + intervalLength/2;
  for(int j=1; j < nClasses; j++)
    classMarkers[j] = classMarkers[j-1] + intervalLength;
  for(int j=1; j <= nClasses+1; j++)
    Borders[j] = Borders[j-1] + intervalLength;
  System.out.println( "\nBorders are" );
  printVector(Borders,nClasses+1 );
  System.out.println( "\nClass markers are" );
  printVector(classMarkers,nClasses );
 double FreqTable[][] = new double[nClasses][2];
 System.out.println(
 "\nEXTENDED TABLE OF ABSOLUTE FREQUENCIES" );
 //reuse
 FreqTable = group(List, Borders, classMarkers );
 meanVarFreqTable(FreqTable);
}


public static void groupM1()
{
//Declaration of data:
```

```
//observed values as a list
//Example 24 pag 12
double[]    List =
{
48.5, 49.2, 51.0, 50.9, 48.7, 50.5, 49.5, 50.8, 50.0, 51.1,
50.7, 51.1, 49.3, 49.1, 50.0, 48.7, 51.2, 49.2, 49.0, 49.3,
49.1, 48.7, 48.6, 49.2, 49.9, 50.1, 50.1, 50.9, 52.4, 50.2,
49.5, 50.9, 51.4, 49.7, 49.8, 50.8, 50.3, 51.8, 50.0, 51.1,
51.3, 50.1, 50.4, 51.1, 49.8, 49.8, 50.1, 50.7, 50.1, 49.9,
49.7, 51.1, 49.7, 49.9, 49.8, 50.5, 49.7, 50.8, 50.7, 50.7,
50.8,  51.5, 51.0
};
System.out.println("\n\n");
System.out.println("FROM DATA LIST TO FREQUENCY TABLE:");
printList(List, "Raw data");
//Borders of the intervals
double[]  Borders = {48.5, 49.5, 50.5, 51.5, 52.5};
double[] classMarkers = {49, 50, 51, 52};
int nClasses = classMarkers.length;
double[][] FreqTable = new double[nClasses][2];
printList(Borders, "Borders of intervals");
printList(classMarkers, "Class markers");
FreqTable = group(List, Borders, classMarkers );
meanVarFreqTable(FreqTable);
}



public static void groupM2( )
{
 //Declaration of data:
 //observed values as a list
 //Example 24 pag 12
 double[]    List =
{
48.5, 49.2, 51.0, 50.9, 48.7, 50.5, 49.5, 50.8, 50.0, 51.1,
50.7, 51.1, 49.3, 49.1, 50.0, 48.7, 51.2, 49.2, 49.0, 49.3,
49.1, 48.7, 48.6, 49.2, 49.9, 50.1, 50.1, 50.9, 52.4, 50.2,
49.5, 50.9, 51.4, 49.7, 49.8, 50.8, 50.3, 51.8, 50.0, 51.1,
```

```
51.3, 50.1, 50.4, 51.1, 49.8, 49.8, 50.1, 50.7, 50.1, 49.9,
49.7, 51.1, 49.7, 49.9, 49.8, 50.5, 49.7, 50.8, 50.7, 50.7,
50.8,  51.5, 51.0
   };
 System.out.println("\n\n");
 System.out.println("FROM DATA LIST TO FREQUENCY TABLE:");
 printList(List, "Raw data");

 //Calculate minimal value of data
 double min = List[0];
 for(int j=1; j < List.length; j++)
    if (List[j] < min)  min = List[j];
 System.out.println( "\nMin value = " + min);


 double infLimit = min;
 //Interval length
 double intervalLength = 1;
 System.out.println("Inferior limit = " + infLimit);
 System.out.println("Interval length = "
  + intervalLength);
 group(List, infLimit, intervalLength);
}

//Data is grouped and processed
public static void groupData( )
{
 //Method 1: all borders and class markers
 // are given by hand.
 groupM1();
 //Method 1: inferior border of first interval and
 //the length  of intervals are given.
 groupM2();
}




//*****************************
```

```
//Pstricks code
//***************************


//Produces the pstricks  code for the bar graph
//of a frequency table
public static void makeCode(double[][] FreqTable)
{
 System.out.println("\nPSTRICKS code for the FreqTable\n");
 int n = FreqTable.length;
 //Width of each column
 double deltaX = 4;
 //The minimum x-coordinate is found
 double Minxx = 1000;
 for ( int i = 0; i < n; i++ )
  if (FreqTable[i][0] < Minxx)
        Minxx =   FreqTable[i][0];
 int Minxxx = (int) (deltaX*Minxx - deltaX);
 //The maximum x-coordinate is found
 double Maxx = 0;
 for ( int i = 0; i < n; i++ )
  if ( FreqTable[i][0] > Maxx)
      Maxx =    FreqTable[i][0];
 int Maxxx = (int) (deltaX*Maxx + deltaX);
 //The maximum y-coordinate is found
 int Maxyy = 0;
 for ( int i = 0; i < n; i++ )
  if (FreqTable[i][1] > Maxyy)
       Maxyy = (int) FreqTable[i][1] + 2;
 //Starting clause
 //x- and y-units.
 //The needed space is defined
 System.out.println("\\begin{center}" +
     "\n\\psset{xunit=0.15,yunit=0.3}" +
 "\n\\begin{pspicture}(" + Minxxx +",0)(" +
 Maxxx+ "," + Maxyy+")");
 //A rectangle is drawn for each entry in the
 //frequency table
```

```
 int lim1,lim2;
 for ( int i = 0; i < n; i++ )
  {
   lim1 = (int) (deltaX*FreqTable[i][0] -deltaX/2);
   lim2 = (int) (lim1 + deltaX);
   System.out.print(
 "\n\\psline("+ lim1 + ",0)" +
     "("+lim1 +"," + FreqTable[i][1]+ ")" +
     "("+lim2 + ","+ FreqTable[i][1]+ ")" +
     "("+lim2 + ","+ "0)"
    );
  }

 //Horizontal axis
 System.out.println("\n\\psline(" +
      Minxxx +",0)("+ Maxxx + ",0)");

 //Class markers (first coordinates) are indicated
 for ( int i = 0; i < n; i++ )
   {
int xposition = (int) (deltaX* FreqTable[i][0]);
int yposition = - 1;
    System.out.print
 (
  "\n\\rput*("+ xposition + "," + yposition+ "){"+
   (int) (FreqTable[i][0])+ "}"
 );
   }
 System.out.println();

 //Labels for heights (second coordinates) are indicated
 for ( int i = 0; i < n; i++ )
  {
   int xposition = (int) (deltaX* FreqTable[i][0]);
   int yposition = (int) (1+ FreqTable[i][1]);
   System.out.print
(
  "\n\\rput*("+ xposition + "," + yposition+ "){" +
      (int) (FreqTable[i][1])+ "}"
```

```
 );
 }

 //End clause
 System.out.println( "\n\\end{pspicture}");
 System.out.println( "\\end{center}");
 }

public static void codePSTricks()
 {
System.out.println("\n\nPSTRICKS CODE FOR A BAR DIAGRAM");
  double[][]  FreqT =
        {{3, 3},
         {4, 3},
         {5, 2},
         {6, 2},
         {7, 1},
         {8, 1}};
 printMatrix(FreqT, "\n\nFREQUENCY  TABLE");
 makeCode(FreqT);
 }




//=================================================
//        BINOMIAL DISTRIBUTION
//=================================================

//The  binomial distribution
//for  n repeats
//probability of head p,
//amplification factor F.
//Arithmetic trickery in action.
private static double[]  binomialDistribution(int n,
          double p, double F  )
{
double[] pBinomial = new double[n+1];
double[] pBinomialF = new double[n+1];
```

```
double[] CoefficientsBinomial = new double[n+1];
double biCoeff = 1;
CoefficientsBinomial[0] = biCoeff;
for(int r = 1; r < n+1; r++)
{
biCoeff = biCoeff * (n+1-r) / r;
CoefficientsBinomial[r] = biCoeff;
}
System.out.println("\nCoefficients ");
printVector(CoefficientsBinomial,n+1);
System.out.println("Sum = " +
     sumVector(CoefficientsBinomial,n+1));

for(int r = 0; r < n+1; r++)
{
pBinomial[r] = CoefficientsBinomial[r]  *
   Math.exp( r* Math.log(p) + (n-r) * Math.log(1-p));
}
System.out.println("\nProbabilities for 0,1,2...");
System.out.println("Paste this vector to Excel" +
           " and make a graphic");
printVector(pBinomial,n+1);
sumVector(pBinomial,n+1);
System.out.println("Sum = "
            +  sumVector(pBinomial,n+1));
for(int k = 0; k < n+1; k++)
 pBinomialF[k] =  F* pBinomial[k];
System.out.println("\nAbsolute frequencies for F = " + F);
printVector(pBinomialF,n+1);
System.out.println("Sum = "  +
           sumVector(pBinomialF,n+1));
return pBinomial;
}

//Probability of closed region (extremes included)
private static double pClosedRegion( int limitInf,
int limitMax,    double[] pBinomial)
{
 double  sum =0;
```

```
 for(int r = limitInf; r <= limitMax; r++)
  sum =  sum +  pBinomial[r];
 return sum;
}

//Detailed computation of the mean
private static void detailedMean(int n,
  double p,
  double[] pBinomial)
{
 System.out.println("\nDetailed mean: xp(x)");
 double  sum =0;
 double term;
 for(int r = 0; r < n+1; r++)
 {
  term =  r *  pBinomial[r];
  System.out.println( r + "    " + term) ;
  sum = sum +term;
 }
 System.out.println( "mean = sum xp(x) = " + sum) ;
 System.out.println( "Expected mean = np =  " + n*p) ;
}

//Detailed computation of the variance
private static void detailedVariance(int n,
  double p,
  double[] pBinomial)
{
 System.out.println("\nDetailed variance: " +
  "(x-mean)*(-mean)*p(x)");
 double  sum =0;
 double term;
 for(int r = 0; r < n+1; r++)
 {
  term =  Math.pow(r- n*p,2) *  pBinomial[r];
  System.out.println( r + "    " +  term) ;
  sum = sum +term;
 }
 System.out.println( "Variance = " +
```

```
"sum (x-mean)*(-mean)*p(x) = " + sum) ;
 System.out.println(
"Expected var = np(1-p) =  " + n*p*(1-p)) ;
 System.out.println( "Deviation = " + Math.sqrt(sum));
}


//Critical value for significance level alpha, one tail
private static void  critical1(  double[] pBinomial,
                                 double alpha )
{
 int numbTails = 1;
 System.out.println("\nLower tail");
 double cumulative = 0;
 double target = alpha;
 int i=-1;
 while( cumulative < target)
 {
  i = i+1;
  cumulative = cumulative + pBinomial[i];
  System.out.println(i + "  cumulative = " + cumulative);
 }
 System.out.println("To cumulate in the lower tail "
    + target
+ ", \nwe need the first " + i  + " Terms.");
 System.out.println("The critical value for alpha =  "
   + alpha
   + ", numb of tails = " + numbTails
   + "\nin the lower tail is "
   + i+ ". All terms below it are extreme.\n");
 System.out.println("Upper tail");
 int n = pBinomial.length;
 i= n;
 cumulative = 0;
 while( cumulative < target)
  {
   i = i-1;
   cumulative = cumulative + pBinomial[i];
   System.out.println(i + "  cumulative = " + cumulative);
```

```
  }

 System.out.println("To cumulate in the upper tail "
    + target
+ ", \nwe need term  " + i  + " and above.");
 System.out.println("The critical value for alpha =  "
   + alpha
   + ", numb of tails = " + numbTails
   + "\nin the upper tail is "
   + i + ". All terms above it are extreme.\n");
}


//Critical value for significance level alpha, two tails
private static  void  critical2(double[] pBinomial,
                           double alpha)
{
 critical1(pBinomial,alpha/2);
}


//Main of binomial distribution
private static void binomialDist(int n, double p, int F)
{
 double[] pBinomial = new double[n+1];
 System.out.println("n= " + n);
 System.out.println("p = " + p);
 System.out.println("F = " + F);
 pBinomial = binomialDistribution(n,p, F);
 detailedMean(n,p,pBinomial);
 detailedVariance(n,p,pBinomial);
 //Probability of a closed region (extremes included)
 int limitInf = 2;
 int limitMax = 4;
 double pR = pClosedRegion(limitInf,limitMax,pBinomial);
 System.out.println( "\nProb of closed region in " +
"within "+ limitInf + "  and " + limitMax +
 ",\nextremes included = " + pR) ;
```

```
}

//Critical values for a given
//significance level
private static void criticalValues(int n, double p,
                            double alpha)
{
 double[] pBinomial = new double[n+1];
 pBinomial = binomialDistribution(n,p, 1);
 System.out.println("Binomial distribution: \n" +
" n = " + n + ", p = " + p + ", " +
"significance level = alpha = " + alpha);
 System.out.println("\nCRITICAL VALUES");
 System.out.println("\nONE TAIL, alpha = " +alpha);
 System.out.println("\nTerms and cumulative values");
 critical1(pBinomial,alpha);
 //critical1(n,p, alpha);
 System.out.println("\nTWO TAILS, alpha = " +alpha);
 System.out.println("\nTerms and cumulative values");
 critical2(pBinomial, alpha);
 //critical2(n,p, alpha);
}

//Binomial distribution
public static void binomialDistribution()
{
 System.out.println("\n\nBINOMIAL DISTRIBUTION ");
 //Example 44 pag 32.
 int n= 6;
 double p = 0.25;
 int F = 35;
 //binomialDist(n,p,F);
 //Inverse problem: find critical values
 double alpha = 0.02;
 criticalValues(n,p,alpha);
}

}//End of main class
```

Problems of Chapter 4

66, pag. 60. A given α decrees associated critical values which define what is normal and what is extreme. So, the probability to commit an error is α-dependent. The probability of an α error is precisely α and that of β-error is $1 - \alpha$.

69, pag. 63. The code that calculates the p-value for the t-distribution follows. The code includes the number of degrees of freedom as a global variable because in that way the code conserves its natural simplicity.

```
//Program F69 pValueT
//Reports  the p- value associated to a given event
//for the t-distribution.
//Combines Simpson's rule
//with a change of scale to
//swiftly calculate the integral
//under the bell in within 0 and  z for any z.
//Style = reuse + global variables.

public class pValueT
{
 private static final double  pi = 3.14159265358979323846;
 private static double dfT;

//Returns numb!
 private static long factorial (long numb)
  {
   if (numb <=1 ) return 1;
else
   return numb * factorial (numb -1);
  }


//Returns the double factorial
private static double doubleFact(double x)
{
 if (x <=0 ) return 1;
else
 return x * doubleFact(x -2);
```

```
 }


//Returns the Gamma function
private static double Gamma(double x)
 {
  int xRounded = (int) x;
  //System.out.println( x + "   " + xRounded);
  if (xRounded == x)
return factorial (xRounded-1);
  else
   {
 return Math.sqrt(pi)* doubleFact(2*xRounded -1) /
          Math.pow(2,xRounded);
   }
 }



//Returns the transformed density function of the
//t distribution with nu d.f.
private static double tFunction(double t, double dfT)
 {
  double nu = dfT;
  double z = t/(1-t*t);
  double  f = ( Gamma((nu+1)/2) /
       ( Math.sqrt(pi*nu)*Gamma(nu/2)) ) *
  Math.pow(1+z*z/nu, -(nu+1)/2 )  * (1+ t*t)/( (1-t*t)*(1-t*t
  return f;
 }


//To use this method for another distribution,
//instead of  tFunction(z), write and make a call for your
//new distribution.
private static double f(double xValue)
 {
  return  tFunction(xValue, dfT);
 }
```

```
//Returns the integral under the function f
//in within 0 and z. Precision=1/N.
//Simpson's rule adapted for reuse.
private static double HalfBodySimpson(double z, long N)
 {
  double h = z/(2*N);
  long m = N;
  double sum = f(0);
  for(int j= 1; j<=m; j++)
 sum = sum +   4* f((2*j-1) * h);
  for(int j= 1; j<m; j++)
 sum = sum +   2* f((2*j) * h);
  sum = sum + f((2*m) * h);
  sum = (h/3)*sum;
  return sum;
 }


//The area in within zero and z of chosen
//distribution is calculated
//for various degrees of precision.
private static  double area(double z)
 {
  z = (Math.sqrt(1+4*z*z)-1)/(2*z);
  int N;
  double  answer;
  int i = 3;
  N = (int) Math.pow(10, i);
  answer = HalfBodySimpson(z,N);
  return answer;
 }

//Returns Psi(e) = p( x < e)
//the cumulative distribution function of z
private static  double psi(double z)
{
 return 0.5 + area(z);
}
```

```
//Returns the p-value(e) = p( x > e)
//the p-value of  z
private static  double pValue(double z)
{
 return 1- psi(z);
}


public static void main(String[] args)
 {
  System.out.println("\nt-distribution");
  //degrees of freedom
  dfT = 7;
  System.out.println("Degrees of freedom = " + dfT);
  double testValue = 1.23456789;
  System.out.println("\nTest value = " + testValue);
  System.out.print("Simpson's rule."+
  "\nArea = p( 0 <  x < " + testValue + ") = ");
  double area = area(testValue);
  System.out.println( area);
  double twoAreas = 2* area;
  System.out.println("Two Areas = " + twoAreas);
  //psi(e) = p( x < e)
  double psi = psi(testValue);
  System.out.println("psi(" + testValue + ") " +
      "= p( x < " + testValue + ") = " +  psi );
  double pValue = pValue(testValue);
  System.out.println("p-value(" +
      testValue + ") = " +
    "p( x > " + testValue + ") = " + pValue);
 }
}
```

70, pag. 63. The code that calculates the p-value for the χ^2-distribution follows:

```
//Program F70 pValueChi2
//Reports  the p- value associated to a given event
```

```
//for the Chi2-distribution.
//Combines Simpson’s rule
//with a change of scale to
//swiftly calculate the integral
//under the bell in within 0 and  z for any z.
//Style = reuse + global variables.

public class pValueChi2
{
 private static final double  pi = 3.14159265358979323846;
 private static double dfChi2;

//Returns numb!
 private static long factorial (long numb)
  {
   if (numb <=1 ) return 1;
else
   return numb * factorial (numb -1);
  }


//Returns the double factorial
private static double doubleFact(double x)
{
 if (x <=0 ) return 1;
else
 return x * doubleFact(x -2);
 }


//Returns the Gamma function
private static double Gamma(double x)
 {
  int xRounded = (int) x;
  //System.out.println( x + "   " + xRounded);
  if (xRounded == x)
return factorial (xRounded-1);
  else
   {
```

```
 return Math.sqrt(pi)* doubleFact(2*xRounded -1) /
          Math.pow(2,xRounded);
   }
 }

//Returns the transformed density function of the
//chi2 distribution with nu d.f.
private static double chi2Function(double t, double nu)
{
double z = t/(1-t*t);
double  f = ( 1 /
       ( Math.pow(2, nu/2)*Gamma(nu/2)) ) *
Math.exp(-z/2)* Math.pow(z, (nu-2)/2) * (1+ t*t)/( (1-t*t)*(1
return f;
}



//To use this method for another distribution,
//instead of  chi2Function(z), write and make a call for your
//new distribution.
private static double f(double xValue)
 {
  return  chi2Function(xValue, dfChi2);
 }

//Returns the integral under the function f
//in within 0 and z. Precision=1/N.
//Simpson's rule adapted for reuse.
private static double HalfBodySimpson(double z, long N)
 {
  double h = z/(2*N);
  long m = N;
  double sum = f(0);
  for(int j= 1; j<=m; j++)
 sum = sum +   4* f((2*j-1) * h);
  for(int j= 1; j<m; j++)
 sum = sum +   2* f((2*j) * h);
  sum = sum + f((2*m) * h);
```

```
  sum = (h/3)*sum;
  return sum;
 }


//The area in within zero and z of chosen
//distribution is calculated
//for various degrees of precision.
private static  double area(double z)
 {
  z = (Math.sqrt(1+4*z*z)-1)/(2*z);
  int N;
  double  answer;
  int i = 3;
  N = (int) Math.pow(10, i);
  answer = HalfBodySimpson(z,N);
  return answer;
 }

//Returns Psi(e) = p( x < e)
//the cumulative distribution function of z
private static  double psiPlus(double z)
{
 return  area(z);
}

//Returns the p-value(e) = p( x > e)
//the p-value of  z
private static  double pValue(double z)
{
 return 1- psiPlus(z);
}


public static void main(String[] args)
 {
  System.out.println("\nChi2-distribution");
  //degrees of freedom
  dfChi2 = 7;
```

```
  System.out.println("Degrees of freedom = " + dfChi2);
  double testValue = 1.23456789;
  System.out.println("\nTest value = " + testValue);
  System.out.print("Simpson's rule."+
  "\nArea = p( 0 <  x < " + testValue + ") = ");
  double area = area(testValue);
  System.out.println( area);
  //psi(e) = p( x < e)
  double psi = psiPlus(testValue);
  System.out.println("psi(" + testValue + ") " +
      "= p( x < " + testValue + ") = " +  psi );
  double pValue = pValue(testValue);
  System.out.println("p-value(" +
      testValue + ") = " +
    "p( x > " + testValue + ") = " + pValue);
 }
}
```

71, pag. 63. The code that calculates the p-value for the F-distribution follows:

```
//Program F71 pValueF
//Reports  the p- value associated to a given event
//for the F-distribution.
//Combines Simpson's rule
//with a change of scale to
//swiftly calculate the integral
//under the bell in within 0 and  z for any z.
//Style = reuse + global variables.

public class pValueF
{
 private static final double  pi = 3.14159265358979323846;
 private static double dfFNum, dfFDen;

//Returns numb!
private static long factorial (long numb)
 {
  if (numb <=1 ) return 1;
```

```
else
  return numb * factorial (numb -1);
 }


//Returns the double factorial
private static double doubleFact(double x)
{
 if (x <=0 ) return 1;
else
 return x * doubleFact(x -2);
 }


//Returns the Gamma function
private static double Gamma(double x)
 {
  int xRounded = (int) x;
  //System.out.println( x + "   " + xRounded);
  if (xRounded == x)
return factorial (xRounded-1);
  else
   {
 return Math.sqrt(pi)* doubleFact(2*xRounded -1) /
          Math.pow(2,xRounded);
   }
 }

//Returns the transformed density function of the
//F distribution with nu1 and nu2 d.f.
private static double FFunction(double t, double nu1, double nu2)
{

double z = t/(1-t*t);
double  f = ( Gamma((nu1+nu2)/2) /
       ( Gamma(nu1/2) *Gamma(nu2/2)  ) ) *
 Math.pow(nu1, nu1/2) * Math.pow(nu2, nu2/2) *
 Math.pow(z,   nu1/2-1)      /
 Math.pow( nu2+nu1*z, (nu1+nu2)/2)
```

```
 *(1+ t*t)/( (1-t*t)*(1-t*t));
return f;
}



//To use this method for another distribution,
//instead of  FFunction(z), write and make a call for your
//new distribution.
private static double f(double xValue)
 {
  return  FFunction(xValue, dfFNum, dfFDen);
 }

//Returns the integral under the function f
//in within 0 and z. Precision=1/N.
//Simpson's rule adapted for reuse.
private static double HalfBodySimpson(double z, long N)
 {
  double h = z/(2*N);
  long m = N;
  double sum = f(0);
  for(int j= 1; j<=m; j++)
 sum = sum +   4* f((2*j-1) * h);
  for(int j= 1; j<m; j++)
 sum = sum +   2* f((2*j) * h);
  sum = sum + f((2*m) * h);
  sum = (h/3)*sum;
  return sum;
 }



//The area in within zero and z of chosen
//distribution is calculated
//for various degrees of precision.
private static  double area(double z)
 {
  z = (Math.sqrt(1+4*z*z)-1)/(2*z);
  int N;
```

```
  double  answer;
  int i = 3;
  N = (int) Math.pow(10, i);
  answer = HalfBodySimpson(z,N);
  return answer;
 }

//Returns Psi(e) = p( x < e)
//the cumulative distribution function of z
private static  double psiPlus(double z)
{
 return  area(z);
}

//Returns the p-value(e) = p( x > e)
//the p-value of  z
private static  double pValue(double z)
{
 return 1- psiPlus(z);
}


public static void main(String[] args)
 {
  System.out.println("\nF-distribution");
  //degrees of freedom
  dfFNum = 7;
  dfFDen = 9;
  System.out.println("Degrees of freedom of numerator = "
  +  dfFNum);
  System.out.println("Degrees of freedom of denominator= "
  + dfFDen);

  double testValue = 1.23456789;
  System.out.println("\nTest value = " + testValue);
  System.out.print("Simpson's rule."+
  "\nArea = p( 0 <  x < " + testValue + ") = ");
  double area = area(testValue);
  System.out.println( area);
```

```
  //psi(e) = p( x < e)
  double psi = psiPlus(testValue);
  System.out.println("psi(" + testValue + ") " +
      "= p( x < " + testValue + ") = " +  psi );
  double pValue = pValue(testValue);
  System.out.println("p-value(" +
      testValue + ") = " +
    "p( x > " + testValue + ") = " + pValue);
 }
}
```

84, pag. 80. The code that calculates p-values for the z, t and χ^2 distributions:

```
//Program F84 pValues3
//Reports   p- value
//for the z, t and chi2 distributions.
//Combines Simpson's rule
//with a change of scale.
//Code is recast in the language of abstract classes.
//This is evolution to the full:
//one must get powerful but with a complexity that
//is perfect  to resolve specific problems.


public class pValues3
{

 //Outer class is instantiated
 static pValues3 a = new pValues3();

 //This is constant pi
 private static final double  pi = 3.14159265358979323846;


//====== Methods of general use=========

//Returns numb!
```

```
private static long factorial (long numb)
 {
  if (numb <=1 ) return 1;
else
  return numb * factorial (numb -1);
 }


//Returns the double factorial
private static double doubleFact(double x)
{
if (x <=0 ) return 1;
else
 return x * doubleFact(x -2);
}


//Returns the Gamma function
private static double Gamma(double x)
{
 int xRounded = (int) x;
 //System.out.println( x + "   " + xRounded);
 if (xRounded == x)
return factorial (xRounded-1);
 else
  {
 return Math.sqrt(pi)* doubleFact(2*xRounded -1) /
          Math.pow(2,xRounded);
  }
}



//======Inner top class with commonalities=========
//The methods that eventually can be useful
//to other distributions come here such as they are.
//The methods that are privative of each distribution
//are defined here as abstract
//and implemented in  specific classes below.
```

```
abstract class pValueAbstract
{
 //Degrees of freedom of various distributions
 double dfT;
 double dfChi2;

//Acquisition of data (abstract declaration)
 pValueAbstract(double a, double b)
 {
  dfT = a;
  dfChi2 = b;
 }

 //The function that defines z is replaced
 //by an abstract type. It is not implemented,
 //so we can implement other functions too and
 //process them with this very same machinery.
 abstract double Function(double t);


//To use this method for another distribution,
//instead of  Function(z), write and make a call for your
//new distribution.
double f(double xValue)
 {
  return  Function(xValue);
 }

//Returns the integral under the function f
//in within 0 and z. Precision=1/N.
//Simpson's rule adapted for reuse.
double HalfBodySimpson(double z, long N)
 {
  double h = z/(2*N);
  long m = N;
  double sum = f(0);
  for(int j= 1; j<=m; j++)
 sum = sum +   4* f((2*j-1) * h);
  for(int j= 1; j<m; j++)
```

```
 sum = sum +   2* f((2*j) * h);
  sum = sum + f((2*m) * h);
  sum = (h/3)*sum;
  return sum;
 }


//The area in within zero and z of chosen
//distribution is calculated
//for various degrees of precision.
double area(double z)
 {
  z = (Math.sqrt(1+4*z*z)-1)/(2*z);
  int N;
  double  answer;
  int i = 3;
  N = (int) Math.pow(10, i);
  answer = HalfBodySimpson(z,N);
  return answer;
 }

//Returns Psi(e) = p( x < e)
//the cumulative distribution function of e.
//This is abstract because it depends
//on the distribution, it must be
//implemented in the corresponding classes.
abstract double psi(double e);


//Returns the p-value(e) = p( x > e)
//Local implementation of the p-value of  e
double pValue(double e)
{
 return 1- psi(e);
}


//Local manager
void executor(double testValue)
```

```
{
 System.out.println("\nTest value = " + testValue);
 System.out.print("Simpson's rule."+
  "\nArea = p( 0 <  x < " + testValue + ") = ");
 double area =  area(testValue);
 System.out.println( area);
 //psi(e) = p( x < e)
 double psi = psi(testValue);
 System.out.println("psi(" + testValue + ") " +
    "= p( x < " + testValue + ") = " +  psi );
 double pValue = pValue(testValue);
 System.out.println("p-value(" +
      testValue + ") = " +
    "p( x > " + testValue + ") = " + pValue);
}

}//end of class pValueAbstract


//======Particular classes========


//The z class.
//Definition of the z density function
//and exclusive to it methods.
//It is an specification of the pValueAbstract class.
class pVzClass extends pValueAbstract
{

 //Constructor: this is the first thing to do
 pVzClass(double dfT, double dfChi2)
 {
  //Coordination with the top class
  super(dfT, dfChi2);
  System.out.println("\nz distribution");
 }

//Returns Psi(e) = p( x < e)
//the cumulative distribution function of e
```

```
double psi(double e)
{
 return 0.5 + area(e);
}

//Function is implemented here.
//Returns the transformed density function of the
//standardized normal distribution.
double Function(double t)
{
 double z = t/(1-t*t);
 double  f = 1 /Math.pow(2*pi, 0.5)*Math.exp(-z*z/2)*
             (1+ t*t)/( (1-t*t)*(1-t*t));
 return f;
}

}//end of class pVzClass


//The t class
//Definition of the t density function
//and exclusive to it methods.
//It is an specification of the pValueAbstract class.
class pVtClass extends pValueAbstract
{

//Constructor: this is the first thing to do
pVtClass(double dfT,double dfChi2)
{
 //Coordination with the top class
 super(dfT,dfChi2);
 System.out.println("\nt distribution");
 System.out.println("t degrees of freedom = " + dfT);
}

//Returns Psi(e) = p( x < e)
//the cumulative distribution function of e
double psi(double e)
{
```

```
 return 0.5 + area(e);
}

//Function is implemented here.
//Returns the transformed density function of the
//t distribution with dfT d.f.
double Function(double t)
{
 double nu = dfT;
 double z = t/(1-t*t);
 double  f = ( Gamma((nu+1)/2) /
       ( Math.sqrt(pi*nu)*Gamma(nu/2)) ) *
 Math.pow(1+z*z/nu, -(nu+1)/2 )  * (1+ t*t)/( (1-t*t)*(1-t*t)
 return f;
}

}//end of class pVtClass



//The chi2 class
//Definition of the Chi2 density function
//and exclusive to it methods.
//It is an specification of the pValueAbstract class.
class pVChi2Class extends pValueAbstract
{

//Constructor: this is the first thing to do
pVChi2Class(double dfT,double dfChi2)
{
 //Coordination with the top class
 super(dfT,dfChi2);
 System.out.println("\nChi2 distribution");
 System.out.println("Chi2 degrees of freedom = " + dfChi2);
}

//Returns Psi(e) = p( x < e)
//the cumulative distribution function of e
double psi(double e)
```

```
{
 return area(e);
}

//Function is implemented here.
//Returns the transformed density function of the
//chi2 distribution with nu d.f.
double Function(double t)
{
 double nu = dfChi2;
 double z = t/(1-t*t);
 double  f = ( 1 /
   ( Math.pow(2, nu/2)*Gamma(nu/2)) )
   * Math.exp(-z/2)* Math.pow(z, (nu-2)/2)
   * (1+ t*t)/( (1-t*t)*(1-t*t));
 return f;
}


}//end of class pVChi2Class


//General manager:
//We  use
//the very same instruction
//to process different distributions
//that can be chosen at will.
public static void main(String args[])
{
 double testValue = 1.23456789;
 //Degrees of freedom of the t distribution
 double dfT = 9;
 //Degrees of freedom of the chi2 distribution
 double dfChi2 = 17;
 //Top class is declared
 pValueAbstract p;

 //Particular class  is instantiated
 pVzClass z = a.new pVzClass(dfT, dfChi2);
```

```
 //Top class is instructed to take
 //the form of the particular class
 p = z;
 //Asked task is made
 p.executor(testValue);

 pVtClass t = a.new pVtClass(dfT, dfChi2);
 p = t;
 p.executor(testValue);

 pVChi2Class chi2 = a.new pVChi2Class(dfT, dfChi2);
 p = chi2;
 p.executor(testValue);
}

}//End of main class
```

85, pag. 80. The code that calculates p-values for the z, t, χ^2 and F distributions:

```
//Program F85 pValues4
//Reports   p- value
//for the z, t, chi2 and F distributions.
//Combines Simpson's rule
//with a change of scale.
//Code is recast in the language of abstract classes.
//This is evolution to the full:
//one must get powerful but with a complexity that
//is perfect  to resolve specific problems.


public class pValues4
{

 //Outer class is instantiated
 static pValues4 a = new pValues4();

 //This is constant pi
```

```
 private static final double  pi = 3.14159265358979323846;


//====== Methods of general use=========

//Returns numb!
private static long factorial (long numb)
 {
  if (numb <=1 ) return 1;
else
  return numb * factorial (numb -1);
 }


//Returns the double factorial
private static double doubleFact(double x)
{
if (x <=0 ) return 1;
else
 return x * doubleFact(x -2);
}


//Returns the Gamma function
private static double Gamma(double x)
{
 int xRounded = (int) x;
 //System.out.println( x + "   " + xRounded);
 if (xRounded == x)
return factorial (xRounded-1);
 else
  {
 return Math.sqrt(pi)* doubleFact(2*xRounded -1) /
          Math.pow(2,xRounded);
  }
}
```

```
//======Inner top class with commonalities=========
//The methods that eventually can be useful
//to other distributions come here such as they are.
//The methods that are privative of each distribution
//are defined here as abstract
//and implemented in  specific classes below.
abstract class pValueAbstract
{
 //Degrees of freedom of various distributions
 double dfT;
 double dfChi2;
 double dfFNum;
 double dfFDen;

//Acquisition of data (abstract declaration)
 pValueAbstract(double a, double b, double c, double d)
 {
  dfT = a;
  dfChi2 = b;
  dfFNum = c;
  dfFDen = d;
 }

 //The function that defines z is replaced
 //by an abstract type. It is not implemented,
 //so we can implement other functions too and
 //process them with this very same machinery.
 abstract double Function(double t);



//Function might be whatever
double f(double xValue)
 {
  return  Function(xValue);
 }

//Returns the integral under the function f
//in within 0 and z. Precision=1/N.
```

```
//Simpson's rule adapted for reuse.
double HalfBodySimpson(double z, long N)
 {
  double h = z/(2*N);
  long m = N;
  double sum = f(0);
  for(int j= 1; j<=m; j++)
 sum = sum +   4* f((2*j-1) * h);
  for(int j= 1; j<m; j++)
 sum = sum +   2* f((2*j) * h);
  sum = sum + f((2*m) * h);
  sum = (h/3)*sum;
  return sum;
 }


//The area in within zero and z of chosen
//distribution is calculated
//for various degrees of precision.
double area(double z)
 {
  z = (Math.sqrt(1+4*z*z)-1)/(2*z);
  int N;
  double  answer;
  int i = 3;
  N = (int) Math.pow(10, i);
  answer = HalfBodySimpson(z,N);
  return answer;
 }

//Returns Psi(e) = p( x < e)
//the cumulative distribution function of e.
//This is abstract because it depends
//on the distribution, it must be
//implemented in the corresponding classes.
abstract double psi(double e);


//Returns the p-value(e) = p( x > e)
```

```
//Local implementation of the p-value of  e
double pValue(double e)
{
 return 1- psi(e);
}


//Local manager
void executor(double testValue)
{
 System.out.println("\nTest value = " + testValue);
 System.out.print("Simpson's rule."+
  "\nArea = p( 0 <  x < " + testValue + ") = ");
 double area =  area(testValue);
 System.out.println( area);
 //psi(e) = p( x < e)
 double psi = psi(testValue);
 System.out.println("psi(" + testValue + ") " +
    "= p( x < " + testValue + ") = " +  psi );
 double pValue = pValue(testValue);
 System.out.println("p-value(" +
      testValue + ") = " +
    "p( x > " + testValue + ") = " + pValue);
}

}//end of class pValueAbstract


//======Particular classes========


//The z class.
//Definition of the z density function
//and exclusive to it methods.
//It is an specification of the pValueAbstract class.
class pVzClass extends pValueAbstract
{

 //Constructor: this is the first thing to do
```

```
 pVzClass(double dfT, double dfChi2,
  double dfFNum, double dfFDen)
 {
  //Coordination with the top class
  super(dfT, dfChi2, dfFNum,  dfFDen);
  System.out.println("\nz distribution");
 }

//Returns Psi(e) = p( x < e)
//the cumulative distribution function of e
double psi(double e)
{
 return 0.5 + area(e);
}

//Function is implemented here.
//Returns the transformed density function of the
//standardized normal distribution.
double Function(double t)
{
 double z = t/(1-t*t);
 double  f = 1 /Math.pow(2*pi, 0.5)*Math.exp(-z*z/2)*
             (1+ t*t)/( (1-t*t)*(1-t*t));
 return f;
}

}//end of class pVzClass


//The t class
//Definition of the t density function
//and exclusive to it methods.
//It is an specification of the pValueAbstract class.
class pVtClass extends pValueAbstract
{

//Constructor: this is the first thing to do
pVtClass(double dfT,double dfChi2,
 double dfFNum, double dfFDen)
```

```
{
 //Coordination with the top class
 super(dfT,dfChi2,  dfFNum,  dfFDen);
 System.out.println("\nt distribution");
 System.out.println("t degrees of freedom = " + dfT);
}

//Returns Psi(e) = p( x < e)
//the cumulative distribution function of e
double psi(double e)
{
 return 0.5 + area(e);
}

//Function is implemented here.
//Returns the transformed density function of the
//t distribution with dfT d.f.
double Function(double t)
{
 double nu = dfT;
 double z = t/(1-t*t);
 double  f = ( Gamma((nu+1)/2) /
       ( Math.sqrt(pi*nu)*Gamma(nu/2)) ) *
 Math.pow(1+z*z/nu, -(nu+1)/2 )  * (1+ t*t)/( (1-t*t)*(1-t*t)
 return f;
}

}//end of class pVtClass



//The chi2 class
//Definition of the Chi2 density function
//and exclusive to it methods.
//It is an specification of the pValueAbstract class.
class pVChi2Class extends pValueAbstract
{

//Constructor: this is the first thing to do
```

```
pVChi2Class(double dfT,double dfChi2,
    double dfFNum, double dfFDen)
{
 //Coordination with the top class
 super(dfT,dfChi2, dfFNum,  dfFDen);
 System.out.println("\nChi2 distribution");
 System.out.println("Chi2 degrees of freedom = " + dfChi2);
}

//Returns Psi(e) = p( x < e)
//the cumulative distribution function of e
double psi(double e)
{
 return area(e);
}

//Function is implemented here.
//Returns the transformed density function of the
//chi2 distribution with nu d.f.
double Function(double t)
{
 double nu = dfChi2;
 double z = t/(1-t*t);
 double  f = ( 1 /
   ( Math.pow(2, nu/2)*Gamma(nu/2)) )
   * Math.exp(-z/2)* Math.pow(z, (nu-2)/2)
   * (1+ t*t)/( (1-t*t)*(1-t*t));
 return f;
}


}//end of class pVChi2Class



//The chi2 class
//Definition of the Chi2 density function
//and exclusive to it methods.
//It is an specification of the pValueAbstract class.
```

```
class pVFClass extends pValueAbstract
{

//Constructor: this is the first thing to do
pVFClass(double dfT,double dfChi2,
    double dfFNum, double dfFDen)
{
//Coordination with the top class
super(dfT,dfChi2, dfFNum,  dfFDen);
System.out.println("\nF distribution");
System.out.println("F degrees of freedom: "
           + "Num = " + dfFNum
           + ", Den = " + dfFDen);
}

//Returns Psi(e) = p( x < e)
//the cumulative distribution function of e
double psi(double e)
{
return area(e);
}

//Function is implemented here.
//Returns the transformed density function of the
//F distribution with dfFNum and dfFDen d.f.
double Function(double t)
{
double nu1 = dfFNum;
double nu2 = dfFDen;
double z = t/(1-t*t);
double  f = ( Gamma((nu1+nu2)/2) /
       ( Gamma(nu1/2) *Gamma(nu2/2)  ) ) *
 Math.pow(nu1, nu1/2) * Math.pow(nu2, nu2/2) *
 Math.pow(z,   nu1/2-1)      /
 Math.pow( nu2+nu1*z, (nu1+nu2)/2)
 *(1+ t*t)/( (1-t*t)*(1-t*t));
return f;
}
```

```
}//end of class pVFClass


//General manager:
//We  use
//the very same instruction
//to process different distributions
//that can be chosen at will.
public static void main(String args[])
{
 double testValue = 1.23456789;
 //Degrees of freedom of the t distribution
 double dfT = 9;
 //Degrees of freedom of the chi2 distribution
 double dfChi2 = 17;
 //Degrees of freedom of the F distribution, numerator
 double dfFNum = 4;
 //Degrees of freedom of the F distribution, denominator
 double dfFDen = 32;
 //Top class is declared
 pValueAbstract p;

 //Particular class z is instantiated
 pVzClass z = a.new pVzClass(dfT, dfChi2, dfFNum,  dfFDen);
 //Top class is instructed to take
 //the form of the particular class
 p = z;
 //Asked task is made
 p.executor(testValue);
 //Particular class t is instantiated
 pVtClass t = a.new pVtClass(dfT, dfChi2, dfFNum,  dfFDen);
 p = t;
 p.executor(testValue);
 //Particular class chi2 is instantiated
 pVChi2Class chi2 =
       a.new pVChi2Class(dfT, dfChi2, dfFNum,  dfFDen);
 p = chi2;
 p.executor(testValue);
```

```
 //Particular class F is instantiated
 pVFClass f =
     a.new pVFClass(dfT, dfChi2, dfFNum,  dfFDen);
 p = f;
 p.executor(testValue);

 //Test:
 //p-values of 1.23456789 by Gnumeric:
 //z: 0.10849568277116
 //t: 0.12412629591734
 //chi2: 0.99999991994117
 //F: 0.31591556865765
}

}//End of main class
```

91, pag. 94. Critical values of the t- distribution.

```
//Program F91 critT
//Reports   critical values
//for a standardized normal distribution
//and a given significance level.
//Combines Simpson's rule
//with a change of scale.
//An initial approximation is improved
//by analytical hunting.
//Program is encapsulated in an inner class.
//Style = Entangled OOP.
public class critT
{

//Outer class is instantiated
static critT a = new critT();


//===========Inner class ==========

//Reports   critical values
//for the t-distribution
```

```
class critTInner
{
 double  pi = 3.14159265358979323846;
 double Error;
 double jCritic;
 double dfT;


//Returns numb!
double factorial (long numb)
 {
  if (numb <=1 ) return 1;
 else
  return numb * factorial (numb -1);
 }


//Returns the double factorial
double doubleFact(double x)
 {
  if (x <=0 ) return 1;
else
  return x * doubleFact(x -2);
 }



//Returns the Gamma function
double Gamma(double x)
 {
  int xRounded = (int) x;
  //System.out.println( x + "   " + xRounded);
  if (xRounded == x)
return factorial (xRounded-1);
  else
   {
 return Math.sqrt(pi)* doubleFact(2*xRounded -1) /
          Math.pow(2,xRounded);
}
```

```
 }


//Returns the transformed density function of the
//t distribution with nu d.f.
//nu is a globally defined variable.
double tFunction(double t)
 {
  double nu = dfT;
  double z = t/(1-t*t);
  double  f = ( Gamma((nu+1)/2) /
       ( Math.sqrt(pi*nu)*Gamma(nu/2)) )
       * Math.pow(1+z*z/nu, -(nu+1)/2 )
       * (1+ t*t)/( (1-t*t)*(1-t*t));
  return f;
 }

//To use this method for another distribution,
//instead of  zFunction(z), write and make a call for your
//new distribution.
double f(double z)
 {
  double  f = tFunction(z);
  return f;
 }


//Reports the transformed upper u-critical for
//the given distribution.
//Significance level = alpha.
//Precision=1/N.
//Simpson's rule adapted for reuse.
//The error of approximation is also calculated.
double upperU(double h, double target)
 {
  double r = h/6;
  double sum = 0.5;
  jCritic = 0;
```

```
  for(int j= 0; sum < target; j++)
  {
 sum = sum +
    r*(f( j*h) + 4* f((j+0.5) * h) +  f((j+1) * h));
 jCritic = j+1;
  }
  jCritic = jCritic-1;
  double greatSum = sum -
         r*(f( jCritic*h) + 4* f((jCritic+0.5) * h)
         +  f((jCritic+1) * h));
  Error = target-greatSum;
  double uAprox = (jCritic) *h;
  return uAprox;
 }


//Returns the transformed value z = u/(1-u*u)
double z(double u)
 {
  double z = u/(1-u*u);
  return z;
 }

//The wide of subintervals is defined
double h()
{
  double u = 1;
  int i=3;
  long N = (int) Math.pow(10, i);
  double h = u/N;
  return h;
}

//Reports the upper z-critical for
//Significance level = alpha.
double vCrit(double target)
 {
  double h = h();
  double uCritic =  upperU(h, target);
```

```
  double vCritic = z(uCritic);
  return vCritic;
 }

//=======Analytical hunting==========

//Returns the density function of the t-distribution
double ff(double z)
 {
  double nu = dfT;
  double  f =  ( Gamma((nu+1)/2) /
       ( Math.sqrt(pi*nu)*Gamma(nu/2)) ) *
    Math.pow(1+z*z/nu, -(nu+1)/2 ) ;
  return f;
 }


// Returns F(a,b) = h(f(a)  + 4 * f((a+b)/2)  + f(b))-Error;
double F(double a, double b)
 {
  double h = (b-a)/6;

  double  Fb = h* (  ff(a) + 4 * ff((a+b)/2)
                   + ff(b)  )-Error;
  return Fb;
 }


//Returns the  ratio  F(zCritic,gi)/ ff(gi)
double ratio(double zCritic, double gi)
 {
  double  ratio =  F(zCritic,gi)/ ff(gi) ;
  return ratio;
 }

//Critical value for the upper tail.
//A professional  approximation is found by
//analytical hunting
double hunting(double zCritic )
```

```
{
double h = h();
double ur =  (jCritic+1)*h;
//Right border of the critical interval
double rBorder = z(ur);
double gSubi = rBorder;
double gSubiPlus= 0;
int iterations = 0;
int maxIterations = 30;
boolean more = true;
while (more)
 {
gSubiPlus = gSubi -  ratio(zCritic, gSubi);
iterations = iterations +1;
if (iterations >= maxIterations)  more = false;
if (gSubiPlus == gSubi) more = false;
gSubi = gSubiPlus;
 }
return gSubiPlus;
}

//Critical value for significance level alpha, one tail
double  critical(  double alpha)
{
double target = 1-alpha;
double zCritic = vCrit(target);
zCritic = hunting(zCritic);
return zCritic;
}

//Local manager
void managerT(double alpha, double df )
 {
  dfT = df;
  System.out.println("\nT distribution");
  System.out.println("Critical values for one tail "
    + "\nof the t-distribution with  "
    + dfT + " degrees of freedom"
+ "\nAlpha = Significance level = "
```

```
+ alpha + ".");

  System.out.println("critical value  = +/-"
+  critical(alpha) );
  System.out.println("\nCritical values for two tails "
    +  "\nof the t-distribution with  "
    + dfT + " degrees of freedom"
+ "\nAlpha = Significance level = "
+ alpha + ".");
 double alphaHalf = alpha/2;
 System.out.println("Critical value  = +/-"
           + critical(alphaHalf) );
 }
}

public static void main(String[] args)
{
 //Inner class is instantiated
 critTInner t = a.new critTInner();
 //Significance level
 double alpha = 0.05;
 //Degrees of freedom
 double df = 7;
 //critical t is required
 t.managerT(alpha,df);
}
}
```

92, pag. 94. Critical values of the Chi2 distribution:

```
//Program F92 critChi2

//Reports   critical values
//for a chi-square-distribution
//and a given significance level.
//Combines Simpson's rule
//with a change of scale.
//An initial approximation is improved
//by analytical hunting.
```

```
//Program is encapsulated in an inner class.
//Style = Entangled OOP.
public class critChi2
{

//Outer class is instantiated
static critChi2 a = new critChi2();


//===========Inner class ==========

//Reports   critical values
//for the t-distribution
public class critChi2Inner
{
double  pi = 3.14159265358979323846;
double Error;
double jCritic;
double dfChi2;


//============Style: reuse============

//Returns numb!
double factorial (long numb)
{
 if (numb <=1 ) return 1;
 else
 return numb * factorial (numb -1);
}



//Returns the double factorial
double doubleFact(double x)
{
 if (x <=0 ) return 1;
 else
 return x * doubleFact(x -2);
```

```
}



//Returns the Gamma function
double Gamma(double x)
{
 int xRounded = (int) x;
 //System.out.println( x + "   " + xRounded);
 if (xRounded == x)
return factorial (xRounded-1);
 else
 {
  return Math.sqrt(pi)* doubleFact(2*xRounded -1) /
          Math.pow(2,xRounded);
 }
}


//===============Style: reuse==================

//Returns the transformed density function of the
//chi-square distribution with nu d.f.
//nu is a globally defined variable.
double chiFunction(double t)
{
 double nu = dfChi2;
 double z = t/(1-t*t);
 double  f = (1/(Math.pow(2, nu/2)*Gamma(nu/2)))
             * Math.exp(-z/2)* Math.pow(z, (nu-2)/2)
             * (1+ t*t)/( (1-t*t)*(1-t*t));
 return f;
}



//To use this method for another distribution,
//instead of  zFunction(z), write and make a call for your
```

```
//new distribution.
double f(double z)
 {
  double  f = chiFunction(z);
  return f;
 }



//Reports the transformed upper u-critical for
//the given distribution.
//Significance level = alpha.
//Precision=1/N.
//Simpson's rule adapted for reuse.
//The error of approximation is also calculated.
//Style = Entangled OOP.
double upperU(double h, double target)
{
 double r = h/6;
 double sum = 0;
 jCritic = 0;
 for(int j= 0; sum < target; j++)
  {
 sum = sum +
   r*(f( j*h) + 4* f((j+0.5) * h) +  f((j+1) * h));
 jCritic = j+1;
  }
 jCritic = jCritic-1;
 double greatSum = sum -
       r*(f( jCritic*h) + 4* f((jCritic+0.5) * h)
         +  f((jCritic+1) * h));
 Error = target-greatSum;
 double uAprox = (jCritic) *h;
 return uAprox;
}

//Returns the transformed value z = u/(1-u*u)
double z(double u)
{
```

```
 double z = u/(1-u*u);
 return z;
}

//The wide of subintervals is defined
double h()
{
  double u = 1;
  int i=3;
  long N = (int) Math.pow(10, i);
  double h = u/N;
  return h;
}

//Reports the upper z-critical for
//Significance level = alpha.
double vCrit(double target)
{
 double h = h();
 double uCritic =  upperU(h, target);
 double zCritic = z(uCritic);
 return zCritic;
}


//=========Analytical hunting============

//Returns the density function of the chi2-distribution
double ff(double z)
{
 double nu = dfChi2;
 double  f = (1/(Math.pow(2, nu/2)*Gamma(nu/2)))
   * Math.exp(-z/2)* Math.pow(z, (nu-2)/2);
 return f;
}


// Returns F(a,b) = h(f(a)  + 4 * f((a+b)/2)  + f(b))-Error;
double F(double a, double b)
```

```
 {
  double h = (b-a)/6;

  double  Fb = h* (  ff(a) + 4 * ff((a+b)/2)
                   + ff(b)  )-Error;
  return Fb;
 }


//Returns the  ratio  F(zCritic,gi)/ ff(gi)
double ratio(double zCritic, double gi)
 {
  double  ratio =  F(zCritic,gi)/ ff(gi) ;
  return ratio;
 }

//Critical value for the upper tail.
//A professional  approximation is found by
//analytical hunting
double hunting(double zCritic )
{
double h = h();
double ur =  (jCritic+1)*h;
//Right border of the critical interval
double rBorder = z(ur);
double gSubi = rBorder;
double gSubiPlus= 0;
int iterations = 0;
int maxIterations = 30;
boolean more = true;
while (more)
 {
gSubiPlus = gSubi -  ratio(zCritic, gSubi);
iterations = iterations +1;
if (iterations >= maxIterations)  more = false;
if (gSubiPlus == gSubi) more = false;
gSubi = gSubiPlus;
 }
return gSubiPlus;
```

```
}

//Critical value for significance level alpha, one tail
double  critical(  double alpha)
{
double target = 1-alpha;
double zCritic = vCrit(target);
zCritic = hunting(zCritic);
return zCritic;
}


//Local manager
void managerChi2(double alpha,double nug )
{
 dfChi2 = nug;
 System.out.println("\nChi2 distribution"
+ "\ngl = " + dfChi2);

 System.out.println("Critical values for one tail "
    + "\nof the chi-square- "
+ "distribution.  "

+ "\nAlpha = Significance level = "
+ alpha + ".");

 System.out.println("Lower critical value  = "
+  critical(1-alpha) );
 System.out.println("Upper critical value  = "
+  critical(alpha) );
 System.out.println("\nCritical values for two tails "
    + "\nof the chi-square "
+ "distribution."
+ "\nAlpha = Significance level = "
+ alpha + ".");
 double alphaHalf = alpha/2;
 double Chi2Menos = critical(1-alphaHalf) ;
 System.out.println("Lower critical value  = "
+  Chi2Menos  );
```

```
 double Chi2Mas = critical(alphaHalf) ;
 System.out.println("Upper critical value  = "
+  Chi2Mas  );
}
}


public static void main(String[] args)
{
 //Inner class is instantiated
 critChi2Inner chi2c = a.new critChi2Inner();
 //Significance level
 double alpha = 0.05;
 //Degrees of freedom
 double df = 7;
 //critical chi2 is required
 chi2c.managerChi2(alpha,df);
}
}
```

93, pag. 94. The code that calculates critical values for the F distribution follows:

```
//Program F93 critF
//Reports   critical values
//for the F-distribution
//and a given significance level.
//Combines Simpson's rule
//with a change of scale.
//An initial approximation is improved
//by analytical hunting.
//Program is encapsulated in an inner class.
//Style = Entangled OOP.
public class critF
{

//Outer class is instantiated
```

```
static critF a = new critF();


//===========Inner class ==========

//Reports   critical values
//for the t-distribution
public class critFInner
{
 double  pi = 3.14159265358979323846;
 double Error;
 double jCritic;
 double dfFNum, dfFDen;
 double k;


//****************Style: reuse************


//Returns the beta   function  with   d.f. =
//a(num) and b(den)
double g(double x, double a, double b)
{
 double  g = Math.pow(x, a -1)*
                 (Math.pow(1-x, b -1));
 return g;
}


//Returns the integral under the function f
//in within 0 and u. Precision=1/N.
//Simpson's rule adapted for reuse.
//a and b are degrees of freedom
double HalfBodySimpsong(double u,
double a, double b, long N)
{
 double h = u/(2*N);
 long m = N;
 double sum = g(0,a,b);
```

```
 for(int j= 1; j<=m; j++)
 sum = sum +   4* g((2*j-1) * h,a,b);
 for(int j= 1; j<m; j++)
 sum = sum +   2* g((2*j) * h,a,b);
 sum = sum + g((2*m) * h,a,b);
 sum = (h/3)*sum;
 return sum;
}


//Returns the  beta  function in integral form
//with a and b d.f.
double beta( double a, double b)
{
 double  answer;
 int i = 4;
 int  N = (int) Math.pow(10, i);
 double u=1;
 answer = HalfBodySimpsong(u,a,b,N);
 return answer;
}


//****************F density function*************

//Returns the transformed density function of the
//F distribution with nu1 and nu2 d.f.
double FFunction(double t)
{
 double nu1 = dfFNum;
 double nu2 = dfFDen;
 double z = t/(1-t*t);
 double  f =   k*Math.pow(z, nu1/2-1)
                  * Math.pow(nu2+nu1*z, -(nu1+nu2)/2)
            * (  (1+ t*t)/( (1-t*t)*(1-t*t))  ) ;
 //System.out.println(f);
 return f;
}
```

```
//To use this method for another distribution,
//instead of  FFunction(z), write and make a call for your
//new distribution.
double f(double z)
{
 double  f = FFunction(z);
 return f;
}


//Reports the transformed upper u-critical for
//the given distribution.
//Significance level = alpha.
//Precision=1/N.
//Simpson's rule adapted for reuse.
//The error of approximation is also calculated.
double upperU(double h, double target)
{
 double nu1 = dfFNum;
 double nu2 = dfFDen;
 double a =  Math.pow(nu1, nu1/2);
 double b =  Math.pow(nu2, nu2/2);
 double c = beta(nu1/2,nu2/2);
 k = a* b / c;
 double r = h/6;
 double sum = 0;
 jCritic = 0;
 for(int j= 0; sum < target; j++)
  {
   sum = sum +
      r*(f( j*h) + 4* f((j+0.5) * h) +  f((j+1) * h));
jCritic = j+1;
  }
 jCritic = jCritic-1;
 double greatSum = sum -
           r*(f( jCritic*h) + 4* f((jCritic+0.5) * h)
         +  f((jCritic+1) * h));
 Error = target-greatSum;
```

```
 double uAprox = (jCritic) *h;
 return uAprox;
}

//Returns the transformed value z = u/(1-u*u)
double z(double u)
{
 double z = u/(1-u*u);
 return z;
}

//The wide of subintervals is defined
double h()
{
  double u = 1;
  int i=3;
  long N = (int) Math.pow(10, i);
  double h = u/N;
  return h;
}

//Reports the upper z-critical for
//Significance level = alpha.
double vCrit(double target)
{
 double h = h();
 double uCritic =  upperU(h, target);
 double zCritic = z(uCritic);
 return zCritic;
}


//*****Analytical hunting*******

//Returns the density function of the t-distribution
double ff(double z)
{
 double nu1 = dfFNum;
 double nu2 = dfFDen;
```

```
 double  f = k*Math.pow(z, nu1/2-1)
     * Math.pow(nu2+nu1*z, -(nu1+nu2)/2)   ;
 return f;
}


// Returns F(a,b) = h(f(a)  + 4 * f((a+b)/2)  + f(b))-Error;
double F(double a, double b)
 {
  double h = (b-a)/6;

  double  Fb = h* (  ff(a) + 4 * ff((a+b)/2)
                   + ff(b)  )-Error;
  return Fb;
 }


//Returns the  ratio  F(zCritic,gi)/ ff(gi)
double ratio(double zCritic, double gi)
 {
  double  ratio =  F(zCritic,gi)/ ff(gi) ;
  return ratio;
 }

//Critical value for the upper tail.
//A professional  approximation is found by
//analytical hunting
double hunting(double zCritic )
{
double h = h();
double ur =  (jCritic+1)*h;
//Right border of the critical interval
double rBorder = z(ur);
double gSubi = rBorder;
double gSubiPlus= 0;
int iterations = 0;
int maxIterations = 30;
boolean more = true;
while (more)
```

```
 {
gSubiPlus = gSubi -  ratio(zCritic, gSubi);
iterations = iterations +1;
if (iterations >= maxIterations)  more = false;
if (gSubiPlus == gSubi) more = false;
gSubi = gSubiPlus;
 }
return gSubiPlus;
}

//Critical value for significance level alpha, one tail
double  critical(  double alpha)
{
double target = 1-alpha;
double zCritic = vCrit(target);
zCritic = hunting(zCritic);
return zCritic;
}


//Local manager
void managerF(double alpha, double nua, double nub )
 {
    System.out.println("\nF distribution");
  dfFNum = nua;
  dfFDen = nub;

System.out.println("Critical values for one tail "
    + "\nof the F- "
+ "distribution with dfs   "
+ dfFNum + " and " + dfFDen+ ", "
+ "\nAlpha = Significance level = "
+ alpha + ".");

System.out.println("Lower critical value  = "
+  critical(1-alpha) );
System.out.println("Upper critical value  = "
+  critical(alpha) );
```

```
System.out.println("\nCritical values for two tails "
    + "\nof the f  "
+ "distribution."
+ "\nAlpha = Significance level = "
+ alpha + ".");
double alphaHalf = alpha/2;

System.out.println("Lower critical value  = "
+  critical(1-alphaHalf) );
System.out.println("Upper critical value  = "
+  critical(alphaHalf) );
 }
}


public static void main(String[] args)
{
 //Inner class is instantiated
 critFInner fc = a.new critFInner();
 //Significance level
 double alpha = 0.05;
 //Degrees of freedom numerator
 double dfFNum = 7;
 //Degrees of freedom denominator
 double dfFDen = 17;
 //critical F is required
 fc.managerF(alpha,dfFNum, dfFDen);
}

}
```

94, pag. 94. The next program calculates the critical values for the z, t, $Chi2$ and F distributions.

```
//Program F94 criticValues
//Reports   critical values
//for the z, t, chi2 and F distributions.
```

```
//Combines Simpson’s rule
//with a change of scale.
//Code is recast in the language of abstract classes.
//This is evolution to the full:
//one must get powerful but with a complexity that
//is perfect  to resolve specific problems.


public class criticValues
{

 //Outer class is instantiated
 static criticValues a = new criticValues();

 //This is constant pi
 private static final double  pi = 3.14159265358979323846;


//====== Methods of general use=========

//Returns numb!
private static long factorial (long numb)
 {
  if (numb <=1 ) return 1;
else
  return numb * factorial (numb -1);
 }


//Returns the double factorial
private static double doubleFact(double x)
{
if (x <=0 ) return 1;
else
 return x * doubleFact(x -2);
}


//Returns the Gamma function
```

```
private static double Gamma(double x)
{
 int xRounded = (int) x;
 //System.out.println( x + "   " + xRounded);
 if (xRounded == x)
return factorial (xRounded-1);
 else
  {
 return Math.sqrt(pi)* doubleFact(2*xRounded -1) /
          Math.pow(2,xRounded);
  }
}



//======Inner top class with commonalities=========
//The methods that eventually can be useful
//to other distributions come here such as they are.
//The methods that are privative of each distribution
//are defined here as abstract
//and implemented in  specific classes below.
abstract class crtValueAbstract
{
 //Degrees of freedom of various distributions
 double dfT;
 double dfChi2;
 double dfFNum;
 double dfFDen;
 double sumzero;
 double Error;
 double jCritic;
 double k;

//Constructor
//Acquisition of data (abstract declaration)
 crtValueAbstract(double a, double b, double c, double d)
 {
  dfT = a;
  dfChi2 = b;
```

```
  dfFNum = c;
  dfFDen = d;
 }




//****************Style: reuse*************


//****************F density function*************

//Returns the transformed density function of the
//F distribution with nu1 and nu2 d.f.
abstract double Function(double t);



//To use this method for another distribution,
//instead of  FFunction(z), write and make a call for your
//new distribution.
double f(double z)
{
 double  f = Function(z);
 return f;
}

//Reports the transformed upper u-critical for
//the given distribution.
//Significance level = alpha.
//Precision=1/N.
//Simpson's rule adapted for reuse.
//The error of approximation is also calculated.
double upperU(double h, double target)
{

double r = h/6;
jCritic = 0;
```

```
double sum = sumzero;
for(int j= 0; sum < target; j++)
 {
 sum = sum +
   r*(f( j*h) + 4* f((j+0.5) * h) +  f((j+1) * h));
 jCritic = j+1;
 }
jCritic = jCritic-1;
double greatSum = sum -
        r*(f( jCritic*h) + 4* f((jCritic+0.5) * h)
         +  f((jCritic+1) * h));
Error = target-greatSum;
double uAprox = (jCritic) *h;
return uAprox;
}



//Returns the transformed value z = u/(1-u*u)
double z(double u)
{
 double z = u/(1-u*u);
 return z;
}

//The wide of subintervals is defined
double h()
{
  double u = 1;
  int i=3;
  long N = (int) Math.pow(10, i);
  double h = u/N;
  return h;
}

//Reports the upper z-critical for
//Significance level = alpha.
double vCrit(double target)
{
```

```
 double h = h();
 //System.out.println( "h = " + h);
 double uCritic =  upperU(h, target);
 double zCritic = z(uCritic);
 return zCritic;
}


//*****Analytical hunting*******

//Returns the density function of the distribution
abstract double ff(double z);



// Returns F(a,b) = h(f(a)  + 4 * f((a+b)/2)  + f(b))-Error;
double F(double a, double b)
 {
  double h = (b-a)/6;
  double  Fb = h* (  ff(a) + 4 * ff((a+b)/2)
                   + ff(b)  )-Error;
  return Fb;
 }


//Returns the  ratio  F(zCritic,gi)/ ff(gi)
double ratio(double zCritic, double gi)
 {
  double  ratio =  F(zCritic,gi)/ ff(gi) ;
  return ratio;
 }

//Critical value for the upper tail.
//A professional  approximation is found by
//analytical hunting
double hunting(double zCritic )
{
double h = h();
double ur =  (jCritic+1)*h;
```

```
//Right border of the critical interval
double rBorder = z(ur);
double gSubi = rBorder;
double gSubiPlus= 0;
int iterations = 0;
int maxIterations = 30;
boolean more = true;
while (more)
 {
gSubiPlus = gSubi -  ratio(zCritic, gSubi);
iterations = iterations +1;
if (iterations >= maxIterations)  more = false;
if (gSubiPlus == gSubi) more = false;
gSubi = gSubiPlus;
 }
return gSubiPlus;
}

//Critical value for significance level alpha, one tail
double  critical(  double alpha)
{
double target = 1-alpha;
double vCritic = vCrit(target);
vCritic = hunting(vCritic);
return vCritic;

}

//Local manager
void manager(double alpha)
  {
System.out.println( "\nAlpha = Significance level = "
+ alpha + ".");

System.out.println("Critical values for ONE TAIL = +/-"
+  critical(alpha) );

double alphaHalf = alpha/2;
System.out.println("Critical values for TWO TAILS = +/-"
```

```
            + critical(alphaHalf) );
  }
}//end of class crtValueAbstract



//======Particular classes========

//The z class.
//Definition of the z density function
//and exclusive to it methods.
//It is an specification of the crtValueAbstract class.
class criticZClass extends crtValueAbstract
{

 //Constructor: this is the first thing to do
 criticZClass(double dfT, double dfChi2,
  double dfFNum, double dfFDen)
 {
  //Coordination with the top class
  super(dfT, dfChi2, dfFNum,  dfFDen);
  sumzero = 0.5; //half bell
  System.out.println("\nz distribution");
 }

 //Returns the transformed density function of
 //the standardized normal distribution.
double Function(double t)
 {
  double z = t/(1-t*t);
  double  f = 1 /Math.pow(2*pi, 0.5)*Math.exp(-z*z/2)*
             (1+ t*t)/( (1-t*t)*(1-t*t));
  return f;
 }

//Returns the density function of the Z distribution
double ff(double z)
 {
  double  f =  Math.exp(-z*z/2)/Math.pow(2*pi, 0.5);
```

```
  return f;
 }
}//end of class criticZClass


//The t class
//Definition of the t density function
//and exclusive to it methods.
//It is an specification of the crtValueAbstract class.
class criticTClass extends  crtValueAbstract
{

//Constructor: this is the first thing to do
criticTClass(double dfT,double dfChi2,
 double dfFNum, double dfFDen)
{
 //Coordination with the top class
 super(dfT,dfChi2,  dfFNum,  dfFDen);
 sumzero = 0.5; //half bell
 System.out.println("\nt distribution");
 System.out.println("t degrees of freedom = " + dfT);
}

//Returns the transformed density function of the
//t distribution with nu d.f.
//nu is a globally defined variable.
double Function(double t)
{
double nu = dfT;
double z = t/(1-t*t);
double  f = ( Gamma((nu+1)/2) /
       ( Math.sqrt(pi*nu)*Gamma(nu/2)) )
       * Math.pow(1+z*z/nu, -(nu+1)/2 )
       * (1+ t*t)/( (1-t*t)*(1-t*t));
return f;
}

//Returns the density function of the t-distribution
double ff(double z)
```

```
 {
  double nu = dfT;
  double  f =  ( Gamma((nu+1)/2) /
       ( Math.sqrt(pi*nu)*Gamma(nu/2)) ) *
    Math.pow(1+z*z/nu, -(nu+1)/2 ) ;
  return f;
 }


}//end of class criticTClass



//The chi2 class
//Definition of the Chi2 density function
//and exclusive to it methods.
//It is an specification of the crtValueAbstract class.
class criticChi2Class extends  crtValueAbstract
{

//Constructor: this is the first thing to do
criticChi2Class(double dfT,double dfChi2,
    double dfFNum, double dfFDen)
{
 //Coordination with the top class
 super(dfT,dfChi2, dfFNum,  dfFDen);
 sumzero = 0; //no negative values
 System.out.println("\nChi2 distribution");
 System.out.println("Chi2 degrees of freedom = " + dfChi2);
}

//Returns the transformed density function of the
//chi-square distribution with nu d.f.
//nu is a globally defined variable.
double Function(double t)
{
double nu = dfChi2;
double z = t/(1-t*t);
double  f = (1/(Math.pow(2, nu/2)*Gamma(nu/2)))
```

```
             * Math.exp(-z/2)* Math.pow(z, (nu-2)/2)
             * (1+ t*t)/( (1-t*t)*(1-t*t));
return f;
}


//Returns the density function of the chi2-distribution
double ff(double z)
{
 double nu = dfChi2;
 double  f = (1/(Math.pow(2, nu/2)*Gamma(nu/2)))
   * Math.exp(-z/2)* Math.pow(z, (nu-2)/2);
 return f;
}


//Local manager
void manager(double alpha)
{
 System.out.println("Critical values for ONE TAIL "
    + "\nof the chi-square- "
+  "distribution with df =  "
+ dfChi2
+ "\nAlpha = Significance level = "
+ alpha + ".");

 System.out.println("Lower critical value  = "
+  critical(1-alpha) );
 System.out.println("Upper critical value  = "
+  critical(alpha) );
 System.out.println("\nCritical values for TWO TAILS "
    + "\nof the chi-square "
+  "distribution with df=  "
+ dfChi2
+ "\nAlpha = Significance level = "
+ alpha + ".");
 double alphaHalf = alpha/2;
 double Chi2Menos = critical(1-alphaHalf) ;
 System.out.println("Lower critical value  = "
```

```
+  Chi2Menos  );
 double Chi2Mas = critical(alphaHalf) ;
 System.out.println("Upper critical value  = "
+  Chi2Mas  );
}
}//end of class criticChi2Class



//The F class
//Definition of the F density function
//and exclusive to it methods.
//It is an specification of the crtValueAbstract class.
class criticFClass extends  crtValueAbstract
{

//Constructor: this is the first thing to do
criticFClass(double dfT,double dfChi2,
    double dfFNum, double dfFDen)
{
//Coordination with the top class
super(dfT,dfChi2, dfFNum,  dfFDen);
System.out.println("\nF distribution");
System.out.println("F degrees of freedom: "
           + "Num = " + dfFNum
           + ", Den = " + dfFDen);
}


//Returns the transformed density function of the
//F distribution with dfFNum and dfFDen d.f.
double Function(double t)
{
double nu1 = dfFNum;
double nu2 = dfFDen;
double z = t/(1-t*t);
double  f =   k*Math.pow(z, nu1/2-1)
                  * Math.pow(nu2+nu1*z, -(nu1+nu2)/2)
            * (  (1+ t*t)/( (1-t*t)*(1-t*t))  ) ;
```

```
//System.out.println(f);
return f;
}


//Returns a helping  function  with   d.f. =
//a(num) and b(den)
double g(double x, double a, double b)
{
double  g = Math.pow(x, a -1)*
                (Math.pow(1-x, b -1));
return g;
}


//Returns the integral under the function f
//in within 0 and u. Precision=1/N.
//Simpson's rule adapted for reuse.
//a and b are degrees of freedom
double HalfBodySimpsong(double u,
double a, double b, long N)
{
double h = u/(2*N);
long m = N;
double sum = g(0,a,b);
for(int j= 1; j<=m; j++)
 sum = sum +   4* g((2*j-1) * h,a,b);
for(int j= 1; j<m; j++)
 sum = sum +   2* g((2*j) * h,a,b);
sum = sum + g((2*m) * h,a,b);
sum = (h/3)*sum;
return sum;
}


//Returns the  beta  function in integral form
//with a and b d.f.
double beta( double a, double b)
{
```

```
double  answer;
int i = 4;
int  N = (int) Math.pow(10, i);
double u=1;
answer = HalfBodySimpsong(u,a,b,N);
return answer;
}


//Reports the transformed upper u-critical for
//the given distribution.
//Significance level = alpha.
//Precision=1/N.
//Simpson’s rule adapted for reuse.
//The error of approximation is also calculated.
double upperU(double h, double target)
{
double nu1 = dfFNum;
double nu2 = dfFDen;
double a =  Math.pow(nu1, nu1/2);
double b =  Math.pow(nu2, nu2/2);
double c = beta(nu1/2,nu2/2);
k = a* b / c;
double r = h/6;
double sum = 0;
jCritic = 0;
for(int j= 0; sum < target; j++)
{
 sum = sum +
    r*(f( j*h) + 4* f((j+0.5) * h) +  f((j+1) * h));
jCritic = j+1;
}
jCritic = jCritic-1;
double greatSum = sum -
           r*(f( jCritic*h) + 4* f((jCritic+0.5) * h)
         +  f((jCritic+1) * h));
Error = target-greatSum;
double uAprox = (jCritic) *h;
return uAprox;
```

```
}


//Returns the density function of the t-distribution
double ff(double z)
{
double nu1 = dfFNum;
double nu2 = dfFDen;
double  f = k*Math.pow(z, nu1/2-1)
   * Math.pow(nu2+nu1*z, -(nu1+nu2)/2)   ;
return f;
}

//Local manager
void manager(double alpha )
 {
    System.out.println("\nF distribution");
System.out.println("Critical values for ONE TAIL "
    + "\nof the F- "
+ "distribution with df:  "
+ dfFNum + " and " + dfFDen+ ", "
+ "\nAlpha = Significance level = "
+ alpha + ".");

System.out.println("Lower critical value  = "
+  critical(1-alpha) );
System.out.println("Upper critical value  = "
+  critical(alpha) );

System.out.println("\nCritical values for TWO TAILS "
    + "\nof the f  "
+ "distribution."
+ "\nAlpha = Significance level = "
+ alpha + ".");
double alphaHalf = alpha/2;

System.out.println("Lower critical value  = "
+  critical(1-alphaHalf) );
System.out.println("Upper critical value  = "
```

```
+  critical(alphaHalf) );
 }
}//end of class criticFClass


//General manager:
//We  use
//the very same instruction
//to process different distributions
//that can be chosen at will.
public static void main(String args[])
{
 double alpha = 0.05;
 //Degrees of freedom of the t distribution
 double dfT = 7;
 //Degrees of freedom of the chi2 distribution
 double dfChi2 = 7;
 //Degrees of freedom of the F distribution, numerator
 double dfFNum = 7;
 //Degrees of freedom of the F distribution, denominator
 double dfFDen = 17;
 //Top class is declared
 crtValueAbstract  p;

 //Particular class z is instantiated
 criticZClass zc = a.new criticZClass(dfT, dfChi2, dfFNum,  dfFDen);
 //Top class is instructed to take
 //the form of the particular class
 p = zc;
 //Asked task is made
 p.manager(alpha);
 //Particular class t is instantiated
 criticTClass tc = a.new criticTClass(dfT, dfChi2, dfFNum,  dfFDen);
 p = tc;
 p.manager(alpha);
 //Particular class chi2 is instantiated
 criticChi2Class chi2c =
       a.new criticChi2Class(dfT, dfChi2, dfFNum,  dfFDen);
 p = chi2c;
```

```
 p.manager(alpha);
 //Particular class F is instantiated
 criticFClass fc =
     a.new criticFClass(dfT, dfChi2, dfFNum,  dfFDen);
 p = fc;
 p.manager(alpha);

//Answers by Gnumeric for alpha = 0.05
//z = -1.64485362695147
//t = 2.3646242515929
//Chi2 = 14.0671404493402
//f  2.61429904513332
}

}//End of main class
```

97, pag. 96. Critical and p-values:

```
//Program F97 values
//Reports  critical and  p- values
//for the z, t, chi2 and F distributions.
//Combines Simpson's rule
//with a change of scale.
//Precision is enhanced by analytical hunting.
//Code is recast in the language of abstract classes.
//This is evolution to the full:
//one must get powerful but with a complexity that
//is perfect  to resolve specific problems.


public class values
{

 //Outer class is instantiated
 static values a = new values();

 //This is constant pi
 private static final double  pi = 3.14159265358979323846;
```

```
//====== Methods of general use=========


//Returns numb!
 private static long factorial (long numb)
  {
   if (numb <=1 ) return 1;
  else
   return numb * factorial (numb -1);
  }

//Returns the double factorial
private static double doubleFact(double x)
{
if (x <=0 ) return 1;
else
 return x * doubleFact(x -2);
}


//Returns the Gamma function
private static double Gamma(double x)
{
 int xRounded = (int) x;
 //System.out.println( x + "   " + xRounded);
 if (xRounded == x)
return factorial (xRounded-1);
 else
  {
 return Math.sqrt(pi)* doubleFact(2*xRounded -1) /
          Math.pow(2,xRounded);
  }
}



//======Inner top class with commonalities=========
```

```
//The methods that eventually can be useful
//to other distributions come here such as they are.
//The methods that are privative of each distribution
//are defined here as abstract
//and implemented in  specific classes below.
abstract class ValueAbstract
{
 //Degrees of freedom of various distributions
 double dfT;
 double dfChi2;
 double dfFNum;
 double dfFDen;
 double sumzero;
 double Error;
 double jCritic;
 double k;

//Acquisition of data (abstract declaration)
 ValueAbstract(double a, double b, double c, double d)
 {
  dfT = a;
  dfChi2 = b;
  dfFNum = c;
  dfFDen = d;
 }

 //The density function is of abstract type.
 //It is not implemented,
 //so we can implement other functions too and
 //process them with this very same machinery.
 abstract double f(double t);



//========= p-values==========

//Returns the integral under the function f
//in within 0 and z. Precision=1/N.
//Simpson's rule adapted for reuse.
```

```
double HalfBodySimpson(double z, long N)
 {
  double h = z/(2*N);
  long m = N;
  double sum = f(0);
  for(int j= 1; j<=m; j++)
 sum = sum +   4* f((2*j-1) * h);
  for(int j= 1; j<m; j++)
 sum = sum +   2* f((2*j) * h);
  sum = sum + f((2*m) * h);
  sum = (h/3)*sum;
  return sum;
 }


//The area in within zero and z of chosen
//distribution is calculated
//for various degrees of precision.
double area(double z)
 {
  z = (Math.sqrt(1+4*z*z)-1)/(2*z);
  int N;
  double  answer;
  int i = 3;
  N = (int) Math.pow(10, i);
  answer = HalfBodySimpson(z,N);
  return answer;
 }

//Returns Psi(e) = p( x < e)
//the cumulative distribution function of e.
//This is abstract because it depends
//on the distribution, it must be
//implemented in the corresponding classes.
double psi(double e)
{
return 0.5 + area(e);
}
```

```
//Returns the p-value(e) = p( x > e)
//Local implementation of the p-value of  e
double pValue(double e)
{
 return 1- psi(e);
}


//Local manager
void pExecutor(double testValue)
{
 System.out.println("\nTest value = " + testValue);
 System.out.print("Simpson's rule."+
  "\nArea = p( 0 <  x < " + testValue + ") = ");
 double area =  area(testValue);
 System.out.println( area);
 //psi(e) = p( x < e)
 double psi = psi(testValue);
 System.out.println("psi(" + testValue + ") " +
    "= p( x < " + testValue + ") = " +  psi );
 double pValue = pValue(testValue);
 System.out.println("p-value(" +
      testValue + ") = " +
    "p( x > " + testValue + ") = " + pValue);
}

//========Critical values=====

//Reports the transformed upper u-critical for
//the given distribution.
//Significance level = alpha.
//Precision=1/N.
//Simpson's rule adapted for reuse.
//The error of approximation is also calculated.
double upperU(double h, double target)
{

double r = h/6;
jCritic = 0;
```

```
double sum = sumzero;
for(int j= 0; sum < target; j++)
{
 sum = sum +
 r*(f( j*h) + 4* f((j+0.5) * h) +  f((j+1) * h));
 jCritic = j+1;
}
jCritic = jCritic-1;
double greatSum = sum -
      r*(f( jCritic*h) + 4* f((jCritic+0.5) * h)
         +  f((jCritic+1) * h));
Error = target-greatSum;
double uAprox = (jCritic) *h;
return uAprox;
}



//Returns the transformed value z = u/(1-u*u)
double z(double u)
{
double z = u/(1-u*u);
return z;
}

//The wide of subintervals is defined
double h()
{
double u = 1;
int i=3;
long N = (int) Math.pow(10, i);
double h = u/N;
return h;
}

//Reports the upper z-critical for
//Significance level = alpha.
double vCrit(double target)
{
```

```
double h = h();
//System.out.println( "h = " + h);
double uCritic =  upperU(h, target);
double zCritic = z(uCritic);
return zCritic;
}


//*****Analytical hunting*******

//Returns the density function of the distribution
abstract double ff(double z);



//Returns F(a,b) = h(f(a)  + 4 * f((a+b)/2)  + f(b))-Error;
double F(double a, double b)
{
double h = (b-a)/6;
double  Fb = h* (  ff(a) + 4 * ff((a+b)/2)
                 + ff(b)  )-Error;
return Fb;
}


//Returns the  ratio  F(zCritic,gi)/ ff(gi)
double ratio(double zCritic, double gi)
{
double  ratio =  F(zCritic,gi)/ ff(gi) ;
return ratio;
}

//Critical value for the upper tail.
//A professional  approximation is found by
//analytical hunting
double hunting(double zCritic )
{
double h = h();
double ur =  (jCritic+1)*h;
```

```
//Right border of the critical interval
double rBorder = z(ur);
double gSubi = rBorder;
double gSubiPlus= 0;
int iterations = 0;
int maxIterations = 30;
boolean more = true;
while (more)
{
gSubiPlus = gSubi -  ratio(zCritic, gSubi);
iterations = iterations +1;
if (iterations >= maxIterations)  more = false;
if (gSubiPlus == gSubi) more = false;
gSubi = gSubiPlus;
}
return gSubiPlus;
}

//Critical value for significance level alpha, one tail
double  critical(  double alpha)
{
double target = 1-alpha;
double vCritic = vCrit(target);
vCritic = hunting(vCritic);
return vCritic;

}

//Local manager
void critManager(double alpha)
{
System.out.println( "\nAlpha = Significance level = "
+ alpha + ".");

System.out.println("Critical values for ONE TAIL = +/-"
+  critical(alpha) );

double alphaHalf = alpha/2;
System.out.println("Critical values for TWO TAILS = +/-"
```

```
            + critical(alphaHalf) );
}

}//end of class pValueAbstract


//======Particular classes========


//The z class.
//Definition of the z density function
//and exclusive to it methods.
//It is an specification of the ValueAbstract class.
class VzClass extends ValueAbstract
{

 //Constructor: this is the first thing to do
 VzClass(double dfT, double dfChi2,
  double dfFNum, double dfFDen)
 {
  //Coordination with the top class
  super(dfT, dfChi2, dfFNum,  dfFDen);
  sumzero = 0.5; //half bell
  System.out.println("\n===== z distribution");
 }


//Function f is implemented here.
//Returns the transformed density function of the
//standardized normal distribution.
double f(double t)
{
 double z = t/(1-t*t);
 double  f = 1 /Math.pow(2*pi, 0.5)*Math.exp(-z*z/2)*
             (1+ t*t)/( (1-t*t)*(1-t*t));
 return f;
}
```

```
//Returns the density function of the Z distribution
double ff(double z)
{
 double  f =  Math.exp(-z*z/2)/Math.pow(2*pi, 0.5);
 return f;
}

}//end of class VzClass


//The t class
//Definition of the t density function
//and exclusive to it methods.
//It is an specification of the ValueAbstract class.
class VtClass extends ValueAbstract
{

//Constructor: this is the first thing to do
VtClass(double dfT,double dfChi2,
 double dfFNum, double dfFDen)
{
 //Coordination with the top class
 super(dfT,dfChi2,  dfFNum,  dfFDen);
 sumzero = 0.5; //half bell
 System.out.println("\n===== t distribution");
 System.out.println("t degrees of freedom = " + dfT);
}



//Function f is implemented here.
//Returns the transformed density function of the
//t distribution with dfT d.f.
double f(double t)
{
 double nu = dfT;
 double z = t/(1-t*t);
 double  f = ( Gamma((nu+1)/2) /
       ( Math.sqrt(pi*nu)*Gamma(nu/2)) ) *
```

```
 Math.pow(1+z*z/nu, -(nu+1)/2 )  * (1+ t*t)/( (1-t*t)*(1-t*t)
 return f;
}


//Returns the density function of the t-distribution
double ff(double z)
{
double nu = dfT;
double  f =  ( Gamma((nu+1)/2) /
       ( Math.sqrt(pi*nu)*Gamma(nu/2)) ) *
    Math.pow(1+z*z/nu, -(nu+1)/2 ) ;
return f;
}

}//end of class VtClass



//The chi2 class
//Definition of the Chi2 density function
//and exclusive to it methods.
//It is an specification of the ValueAbstract class.
class VChi2Class extends ValueAbstract
{

//Constructor: this is the first thing to do
VChi2Class(double dfT,double dfChi2,
    double dfFNum, double dfFDen)
{
 //Coordination with the top class
 super(dfT,dfChi2, dfFNum,  dfFDen);
 sumzero = 0; //no negative values
 System.out.println("\n===== Chi2 distribution");
 System.out.println("Chi2 degrees of freedom = " + dfChi2);
}

//Returns Psi(e) = p( x < e)
//the cumulative distribution function of e
```

```
double psi(double e)
{
 return area(e);
}

//Function f is implemented here.
//Returns the transformed density function of the
//chi2 distribution with nu d.f.
double f(double t)
{
 double nu = dfChi2;
 double z = t/(1-t*t);
 double  f = ( 1 /
   ( Math.pow(2, nu/2)*Gamma(nu/2)) )
   * Math.exp(-z/2)* Math.pow(z, (nu-2)/2)
   * (1+ t*t)/( (1-t*t)*(1-t*t));
 return f;
}



//Returns the density function of the chi2-distribution
double ff(double z)
{
double nu = dfChi2;
double  f = (1/(Math.pow(2, nu/2)*Gamma(nu/2)))
 * Math.exp(-z/2)* Math.pow(z, (nu-2)/2);
return f;
}


//Local manager
void manager(double alpha)
{
System.out.println("Critical values for ONE TAIL "
    + "\nof the chi-square- "
+  "distribution with df =  "
+ dfChi2
+ "\nAlpha = Significance level = "
```

```
+ alpha + ".");

System.out.println("Lower critical value  = "
+  critical(1-alpha) );
System.out.println("Upper critical value  = "
+  critical(alpha) );
System.out.println("\nCritical values for TWO TAILS "
    + "\nof the chi-square "
+  "distribution with df=  "
+ dfChi2
+ "\nAlpha = Significance level = "
+ alpha + ".");
double alphaHalf = alpha/2;
double Chi2Menos = critical(1-alphaHalf) ;
System.out.println("Lower critical value  = "
+  Chi2Menos  );
double Chi2Mas = critical(alphaHalf) ;
System.out.println("Upper critical value  = "
+  Chi2Mas  );
}
}//end of class pVChi2Class



//The F class
//Definition of the Chi2 density function
//and exclusive to it methods.
//It is an specification of the ValueAbstract class.
class VFClass extends ValueAbstract
{

//Constructor: this is the first thing to do
VFClass(double dfT,double dfChi2,
    double dfFNum, double dfFDen)
{
//Coordination with the top class
super(dfT,dfChi2, dfFNum,  dfFDen);
sumzero = 0; //no negative values
double nu1 = dfFNum;
```

```
double nu2 = dfFDen;
double a =  Math.pow(nu1, nu1/2);
double b =  Math.pow(nu2, nu2/2);
double c = beta(nu1/2,nu2/2);
k = a* b / c;
System.out.println("\n===== F distribution");
System.out.println("F degrees of freedom: "
           + "Num = " + dfFNum
           + ", Den = " + dfFDen);
}

//Returns Psi(e) = p( x < e)
//the cumulative distribution function of e
double psi(double e)
{
return area(e);
}



//Returns the transformed density function of the
//F distribution with dfFNum and dfFDen d.f.
double f(double t)
{
double nu1 = dfFNum;
double nu2 = dfFDen;
double z = t/(1-t*t);
double  f =   k*Math.pow(z, nu1/2-1)
                  * Math.pow(nu2+nu1*z, -(nu1+nu2)/2)
            * (  (1+ t*t)/( (1-t*t)*(1-t*t))  ) ;
//System.out.println(f);
return f;
}


//Returns a helping  function  with   d.f. =
//a(num) and b(den)
double g(double x, double a, double b)
{
```

```
double  g = Math.pow(x, a -1)*
                (Math.pow(1-x, b -1));
return g;
}


//Returns the integral under the function f
//in within 0 and u. Precision=1/N.
//Simpson's rule adapted for reuse.
//a and b are degrees of freedom
double HalfBodySimpsong(double u,
double a, double b, long N)
{
double h = u/(2*N);
long m = N;
double sum = g(0,a,b);
for(int j= 1; j<=m; j++)
 sum = sum +   4* g((2*j-1) * h,a,b);
for(int j= 1; j<m; j++)
 sum = sum +   2* g((2*j) * h,a,b);
sum = sum + g((2*m) * h,a,b);
sum = (h/3)*sum;
return sum;
}


//Returns the  beta  function in integral form
//with a and b d.f.
double beta( double a, double b)
{
double  answer;
int i = 4;
int  N = (int) Math.pow(10, i);
double u=1;
answer = HalfBodySimpsong(u,a,b,N);
return answer;
}
```

```
//Reports the transformed upper u-critical for
//the given distribution.
//Significance level = alpha.
//Precision=1/N.
//Simpson's rule adapted for reuse.
//The error of approximation is also calculated.
double upperU(double h, double target)
{

double r = h/6;
double sum = 0;
jCritic = 0;
for(int j= 0; sum < target; j++)
{
sum = sum +
  r*(f( j*h) + 4* f((j+0.5) * h) +  f((j+1) * h));
jCritic = j+1;
}
jCritic = jCritic-1;
double greatSum = sum -
           r*(f( jCritic*h) + 4* f((jCritic+0.5) * h)
         +  f((jCritic+1) * h));
Error = target-greatSum;
double uAprox = (jCritic) *h;
return uAprox;
}


//Returns the density function of the t-distribution
double ff(double z)
{
double nu1 = dfFNum;
double nu2 = dfFDen;
double  f = k*Math.pow(z, nu1/2-1)
 * Math.pow(nu2+nu1*z, -(nu1+nu2)/2)   ;
return f;
}

//Local manager
```

```
void manager(double alpha )
{
  System.out.println("\nF distribution");
System.out.println("Critical values for ONE TAIL "
    + "\nof the F- "
+ "distribution with df:  "
+ dfFNum + " and " + dfFDen+ ", "
+ "\nAlpha = Significance level = "
+ alpha + ".");

System.out.println("Lower critical value  = "
+  critical(1-alpha) );
System.out.println("Upper critical value  = "
+  critical(alpha) );

System.out.println("\nCritical values for TWO TAILS "
    + "\nof the f  "
+ "distribution."
+ "\nAlpha = Significance level = "
+ alpha + ".");
double alphaHalf = alpha/2;

System.out.println("Lower critical value  = "
+  critical(1-alphaHalf) );
System.out.println("Upper critical value  = "
+  critical(alphaHalf) );
}
}//end of class pVFClass



//General manager:
//We  use
//the very same instruction
//to process different distributions
//that can be chosen at will.
public static void main(String args[])
{
 double testValue = 1.23456789;
```

```
double alpha = 0.05;
//Degrees of freedom of the t distribution
double dfT = 9;
//Degrees of freedom of the chi2 distribution
double dfChi2 = 17;
//Degrees of freedom of the F distribution, numerator
double dfFNum = 4;
//Degrees of freedom of the F distribution, denominator
double dfFDen = 32;
//Top class is declared
ValueAbstract p;

//Particular class z is instantiated
VzClass z = a.new VzClass(dfT, dfChi2, dfFNum,  dfFDen);
//Top class is instructed to take
//the form of the particular class
p = z;
//p-values are calculated
p.pExecutor(testValue);
//Critical values are calculated
p.critManager(alpha);
//Particular class t is instantiated
VtClass t = a.new VtClass(dfT, dfChi2, dfFNum,  dfFDen);
p = t;
p.pExecutor(testValue);
p.critManager(alpha);
//Particular class chi2 is instantiated
VChi2Class chi2 =
      a.new VChi2Class(dfT, dfChi2, dfFNum,  dfFDen);
p = chi2;
p.pExecutor(testValue);
p.critManager(alpha);
//Particular class F is instantiated
VFClass f =
    a.new VFClass(dfT, dfChi2, dfFNum,  dfFDen);
p = f;
p.pExecutor(testValue);
p.critManager(alpha);
```

```
 //Test:
 //p-values of 1.23456789 by Gnumeric:
 //z: 0.10849568277116
 //t: 0.12412629591734
 //chi2: 0.99999991994117
 //F: 0.31591556865765


//Critical values by Gnumeric for alpha = 0.05
//z = -1.64485362695147
//t = 2.26215716279837
//Chi2 = 27.5871116382753
//f= 2.66843694251984
}

}//End of main class
```

Problems of Chapter 5

109, pag. 161. The first version of StatTrek, our toolBox for statistics, that includes the top 12 experiments follows.

```
//Program F109 StatTrek1
/*

 StatTrek version 1

 This is a tool box of methods specially suited
 to support teachers of an elementary course in
 statistics. We go along the companion book
 Elementary Statistics Vol I version 7 (in Spanish).

 The basics of Java can be learned from

 Java for the study of evolution
 Vol V, the scientific method with Java.
 Actually, this program is the response given by
 the Author to   challenge 349 on page 375
```

of that volume.

Aforementioned material can be down loaded from www.evoljava.com

License: you are invited to use, reuse and misuse this software even to make money if only you can help the formation of a strong and mature scientific community.

This is a tool box of methods to calculate
 a) some statistics of a list of data,
 b) the pstricks code for a bar diagram,
 c) the binomial distribution and its critical values,
 d) the 12 most popular experiments.

How to use this program:

1. Append it to any project and run it.
2. This program is a tool box because its main method can be filled with any allowed method taken from the demo.
3. Most relevant allowed methods can be listed in the main one but only one must be activated, all others must be muted.
4. To silence one method, append // as a prefix. To activate a method, delete the prefix //.
5. For a given task, choose the best suited method among the diverse variants.
6. To run a method with your own data, modify the inbuilt data according to your needs. Or much better: make a copy of the needed method, arrange troubles with duplicated names and modify the copy.
7. This first version has just one aim: To implement fundamental tasks as transparently as possible. So, it looks rather like an array of independent, non interacting methods.

```
 8. To inquire about unknown terms, drag the mouse over
    the term and wait until Eclipse displays the
    corresponding documentation.
 9. To go   from the name of a
    method  to its implementation:
     Pose the cursor at the end of the line
     with the name of the method and press the space bar,
     next press F3: you will be transported to the
     implementation of the method.
 10. To mark a specific site in the program:
     -Activate line numbering.
     -Pose the cursor over the line you want to mark.
     -Go to the corresponding number at the left margin
      of the Editor Window and right click.
     -Choose the Add Bookmark menu and punch it.
     -A blue marker will appear in the right margin of the
      Editor Window. By clicking over it, one can
      return to the desired place.
      One can also play with the menu
      Window -> ShowView -> Outline.
      A list with all methods will appear:
      click over any one and you will
      transported to the corresponding place
      along the text of the program.

*/



public class StatTrek1
{

 //The outer application is instantiated
 private static StatTrek1 a = new StatTrek1();

//This is constant pi
private static final double  pi = 3.14159265358979323846;

//Symbolic initialization that can be modified
```

```
//Significance level
private static double alpha;
//Degrees of freedom of the t distribution
private static double dfT;
//Degrees of freedom of the chi2 distribution
private static double dfChi2;
//Degrees of freedom of the F distribution, numerator
private static double dfFNum;
//Degrees of freedom of the F distribution, denominator
private static double dfFDen ;
private static double k;

 //Top abstract class is declared
 private static  ValueAbstract p;
 //Particular class z is instantiated
 private static zClass z =  a.new zClass();
 private static tClass t =  a.new tClass();
 private static chi2Class chi2 =  a.new chi2Class();
 private static FClass F =  a.new FClass();
//============================
//MAIN--MAIN--MAIN--MAIN--MAIN
//============================

 public static void main(String[] args)
 {
  demo();
  //meanVarList();
  //meanVarFreqTable();
  //meanVarCumTable();
  //groupData();
  //codePSTricks();
  //binomialDistribution();
  //partitionZ();
  //partitionX();

  //zTestOneMean();
  //zTestTwoMeans();
  //tTestOneMean();
  //zTestOneProp();
```

```
  //zTestTwoProp();
  //chi2TestOneVar();
  //FTestTwoVariances( );

  //Bivariate tests about means
  //tTestTwoMeansEqualVar();
  //tTestTwoMeansDiffVar();
  //tTestFullCompTwoMeans();
  //tTestFullCompTwoMeansLists();
  //tTestFullCompTwoMeansFT();
  //fullCompTwoMeansCF();

  //tTestPairedData();

  //factorIndependence();
  //chi2TestPropHomogeneity();

 }

 //Demo
 public static void demo()
  {
   meanVarList();
   meanVarFreqTable();
   meanVarCumTable();
   groupData();
   codePSTricks();
   binomialDistribution();
   partitionZ();
   partitionX();

   zTestOneMean();
   zTestTwoMeans();
   tTestOneMean();
   zTestOneProp();
   zTestTwoProp();
   chi2TestOneVar();
   FTestTwoVariances( );
```

```
   //Bivariate tests about means
   tTestTwoMeansEqualVar();
   tTestTwoMeansDiffVar();
   tTestFullCompTwoMeans();
   tTestFullCompTwoMeansLists();
   tTestFullCompTwoMeansFT();
   fullCompTwoMeansCF();

   tTestPairedData();

   factorIndependence();
   chi2TestPropHomogeneity();
  }

//================================================
//               IMPLEMENTATIONS
//================================================




//================================================
//      Methods of general use
//================================================

//Prints a separator
private static void separator()
 {
  System.out.println( "\n========*=========\n");
 }


//Prints a list and its title
private static void printList(double DataVector[],
                              String title)
{
 System.out.println( "\n " + title );
 int nData = DataVector.length;
 boolean go = true;
```

```
 int nCols = 10;
 int nRows = 9;
 int counter = 0;
 for (int i = 0; (i < nRows) & go; i++)
  {
   for (int j = 0; (j < nCols) & go; j++)
    {
     System.out.print( DataVector[i*10 + j] + "   ");
     if (counter >= nData-1) go = false;
     counter = counter+1;
    }
  System.out.println( );
 }
 System.out.print("\nNumber of data = " +nData + "\n");
}


//Prints n data in vector
private static void printVector(double Vect[], int n)
{
 //System.out.println( "Data are" );
 for ( int i = 0; i < n; i++ )
   System.out.println(i + "\t" +  Vect[ i ]  );
}


//Sums data in vector with real numbers
private static double sumVector(double Vect[], int n)
{
 double sum = 0;
 for ( int i = 0; i < n; i++ )
 sum = sum + Vect[i];
 return sum;
}



//Prints  a matrix and is title
private static void printMatrix(double Matrix[][],
```

```
                        String title)
{
 System.out.println( title );
int m = Matrix.length;
 for ( int i = 0; i < m; i++ )
 System.out.println( Matrix[ i ][0]+
        "\t "+ Matrix[ i ][1]);
}


//Documentation is printed
private static void explanation()
{
 System.out.println( "\nCoeff around 0.01, " +
                     "homogeneous population." +
                     "\nCoeff around 0.1, " +
                     " +/- homogeneous population."+
                     "\nCoeff around 0.5, " +
                     "  heterogenous population.");
}


//================================================
//       MEAN AND VAR O A LIST OF DATA
//================================================



 //======== Inner class definition=======

 //This inner class defines a new type: Table.
 //It converts array F (two dimensional) into
 //a class or object, ready for reuse.
 private  class Table
 {
  //The class Table fundamentally has three elements:
  //a matrix with two columns and its length,
  //which is the number of rows and
  //the titles of columns.
```

```
 int length;
 String[] Titles = { "x",  "x2 = x^2"};
 double M[][] = new double[2000][2];

 //The input of Table is defined here
 //by means of an initialization procedure
 //that is called constructor.
 //The natural input is the raw data,
 //which is consigned in a one dimensional array.
 Table(double[] V )
 {
  length = V.length;
  //The matrix M of Table is initialized
  //Raw data go in the first column.
  //Squares in the second.
  for(int i = 0; i < length; i++)
  {
M[i][0] = V[i];
M[i][1] = V[i]*V[i];
  }
 }

//Data of column k are summed.
//The input is a table, the output is of type double.
private  double sum(Table  T, int k)
 {
  double sum = 0;
  for(int i = 0; i < T.length; i++)
 sum  = sum +  T.M[i][k];
  return sum;
 }

//Mean of data.
private  double mean(Table  T)
 {
  double sumx = T.sum(T,0);
  int n = T.length;
  double mean = sumx/ n;
  return mean;
```

```
  }

 //Var of data
 private  double var(Table  T)
  {
   double sumx = T.sum(T,0);
   double sumx2 = T.sum(T,1);
   int n = T.length;
   double var = (n * sumx2 - sumx*sumx)/ (n*( n-1));
   return var;

  }

 private  void printTable(Table T)
  {
   System.out.printf("%n %10s  %11s",
              T.Titles [0], T.Titles [1]);
   for ( int i = 0; i < T.length; i++ )
  System.out.printf("%n %10.3f   %10.3f ",
        T.M[i][0], T.M[i][1] );
   System.out.println( );
  }
}//end of inner class


//=== Methods of the outer class ===


//This is the core of the main class
//Here, commands are given to execute the overall task.
private static void meanVarVector(double[] V)
{
 //A table is declared
 Table T =  a.new   Table(V);
 T.printTable(T);
 double sumx = T.sum(T,0);
 int n = T.length;
 double mean = sumx/ n;
 double sumx2 = T.sum(T,1);
```

```
 System.out.printf("%n Sum %6.3f   %10.3f ",
                 sumx,        sumx2 );
 System.out.print("\n \n");
 System.out.println("n = number of data = " + n);

 System.out.println("\nsumx = " + sumx);
 System.out.println("Mean = sumx / n");
 System.out.println("Mean = " + mean);
 System.out.println("sumx2 = " + sumx2);
 System.out.println("variance = " +
   "(n * sumx2 - sumx*sumx)/ (n*( n-1))");
 double var = (n * sumx2 - sumx*sumx)/ (n*( n-1));
 System.out.println( "Variance = "+ var);
 double deviation = Math.pow( var, 0.5);
 System.out.println( "Deviation = "+ deviation);
 double coeffVar =  deviation/mean;
 System.out.println( "Coefficient of variation " +
                 "= deviation/mean");
 System.out.println( "Coefficient of variation = "
                                 + coeffVar);
 explanation();
}

//A list of data  is  processed
private static void meanVarList( )
{
  System.out.println("\n\n");
  System.out.println( "MEAN AND VAR OF A LIST OF DATA");
  //Example 19 page 9.
  double  List[]  = {5, 3, 4.5, 2};
  //Prints list and title
  printList(List,  "Raw data");
  System.out.println( "\nTABLE WITH x AND ITS SQUARE");
  meanVarVector(List);
  separator();
}

//=================================================
// MEAN AND VAR OF A TABLE OF ABSOLUTE FREQUENCIES
```

```
//=================================================


//======== Inner class definition=======

//This inner class defines a new type: Table.
//It converts array F (two dimensional) into
//a class or object, ready for reuse.
private  class FreqTable
 {
//The class Table has two main elements:
//1) The titles of  the columns, whose length
//gives the number of columns.
//2) A matrix, whose length is the number of rows.

String[] Titles = { "x",  "F",   "xF", "x2F"};
double M[][] = new double[2000][Titles.length];
int nRows;
//The input of Table is defined here
//by means of an initialization procedure
//that is called constructor.
//The natural input is the frequency table
//which is consigned in a matrix FT with 2 columns.
FreqTable(double FT[][] )
 {
   nRows = FT.length;
   //The matrix M of Table is initialized
   //Raw data go in   columns 0 and 1.
   //Squares in the second.
   for(int i = 0; i < nRows; i++)
   {
 M[i][0] = FT[i][0];
 M[i][1] = FT[i][1];
 M[i][2] = M[i][0] * M[i][1];
 M[i][3] = M[i][0] * M[i][2];
   }
  }

//Data of column k are summed.
```

```
//The input is a table, the output is of type double.
private  double sum(FreqTable  T, int k)
 {
  double sum = 0;
  for(int i = 0; i < T.nRows; i++)
  sum  = sum +  T.M[i][k];
  return sum;
 }


 //Mean of data.
 private  double mean(FreqTable  T)
  {
   double n = T.sum(T,1);
   double sumxF = T.sum(T,2);
   double mean = sumxF/ n;
   return mean;
  }

 //Var of data
 private  double var(FreqTable  T)
  {
   double n = T.sum(T,1);
   double sumxF = T.sum(T,2);
   double sumx2F = T.sum(T,3);
   double Sxx  = sumx2F -    Math.pow(sumxF, 2)/n;
   double var = Sxx / (n-1);
   return var;
  }

private  void printTable(FreqTable T)
 {
System.out.println("\nExtended matrix to " +
                "calculate variance:");
  System.out.printf("%n %10s %10s %10s %10s",
            T.Titles [0], T.Titles [1],
             T.Titles [2],T.Titles [3]);
    for ( int i = 0; i < T.nRows; i++ )
 System.out.printf(
```

```
"%n %10.3f %10.3f %10.3f %10.3f ",
T.M[i][0], T.M[i][1], T.M[i][2], T.M[i][3] );
  System.out.println( );
 }
}//end of inner class


//=== Methods of the outer class ===


// Mean and variance of a frequency table
private static void meanVarFreqTable(double FreqTable[][])
{
 //A table is declared
 FreqTable T =  a.new   FreqTable(FreqTable);
 T.printTable(T);
 double n = T.sum(T,1);
 double sumxF = T.sum(T,2);
 double sumx2F = T.sum(T,3);
 System.out.printf("Sum %18.3f %10.3f %10.3f ",
             n,   sumxF,        sumx2F );
 double mean = sumxF/ n;
 System.out.print(
  "\n\n n = Number of  objects = sum F = "+ (int) n);
 System.out.printf( "\n sum xF  = %10.3f ", sumxF);
 System.out.printf( "\n sum x2F  = %10.3f ",  sumx2F);
 System.out.printf( "\n Mean = sum xF / n = %10.3f ", mean);
 double Sxx  = sumx2F -    Math.pow(sumxF, 2)/n;
 System.out.printf(
 "\n Sxx = sum x2F - (sumxF)^2 /n = %10.3f ", Sxx);
 double var = Sxx / (n-1);
 System.out.printf(
 "\n Variance =  Sxx/(n-1) = %10.3f ", var);
 double deviation = Math.sqrt(var);
 System.out.printf(
 "\n Deviation =  %10.3f ", deviation);
 double c = deviation/ mean;
 System.out.printf(
   "\n Coefficient of variation =  %10.3f \n", c);
```

```
 explanation();
}


//A Frequency Table is processed
private static void meanVarFreqTable()
{
 System.out.println("\n\n");
 System.out.println("\n\nMEAN-VAR OF A FREQUENCY TABLE");
 //Frequency table: first coordinate = class marker
 //second coordinate = frequency

 //Example 20, pag 10
 double  FreqT[][] =
        {{4, 3},
         {7, 6},
         {10, 10},
         {13, 15},
         {16, 4},
         {19, 2}};

 printMatrix(FreqT, "\n Raw data");
 meanVarFreqTable(FreqT);
 separator();
}


//=================================================
//MEAN AND VAR OF A TABLE OF CUMULATIVE FREQUENCIES
//=================================================


private static void meanVarCumTable(double[][] cumFreqT)
{
 printMatrix(cumFreqT, "\n Raw data");
 //Absolute frequencies are calculated
 //and kept in FreqT, a matrix with two columns.
 double FreqT[][] = new double[cumFreqT.length][2];
```

```
 FreqT[0] = cumFreqT[0];
 int lx = cumFreqT.length;
 for(int i = 1; i < lx;  i++)
 {
  FreqT[i][0] = cumFreqT[i][0];
  FreqT[i][1] = cumFreqT[i][1]-cumFreqT[i-1][1];
 }
 System.out.println("\n\n");
 System.out.println(
 "\n MEAN-VAR OF ASSOCIATED  FREQUENCY TABLE");
  meanVarFreqTable(FreqT);//reuse
 }

//A cumulative frequency table is processed
public static void meanVarCumTable()
{
 System.out.println(
 "MEAN-VAR OF A CUMULATIVE FREQUENCY TABLE");
 //Cumulative table: first coordinate = class marker
 //second coordinate = cumulative frequency

 //Problem 1 of version 1,  pag 42.
 double  cumFreqT[][] =
        {{32,7},
         {40,12},
         {45,17},
         {50,45},
         {55,52},
         {60,58}};
  meanVarCumTable(cumFreqT);
  separator();
}


//=================================================
// GROUPING OF DATA IN INTERVALS + MEAN & VAR
//=================================================
```

```
//List is the input, a frequency table is the output
private static double[][] group(double List[],
        double Borders[], double classMarkers[]  )
{
 int nClasses = classMarkers.length;
 double FreqTable[][] = new double[nClasses][2];
 int nData = List.length;

 for(int j=0; j < nClasses; j++)
  {
FreqTable[j][0] = classMarkers[j];
FreqTable[j][1] = 0;
  }
 for(int i=0; i< nData; i++)
  {
   for(int j=0; j < nClasses; j++)
 {
  if((Borders[j] <= List[i]) & (List[i] < Borders[j+1]))
  FreqTable[j][1] =FreqTable[j][1] +1;
}

 }

 return FreqTable;
}




//Data is grouped into a frequency table.
//The borders and class marker of each interval
//is calculated.
//Final output = Mean and var of the freq table.
private static void group(double List[],
double infLimit, double intervalLength)
{
  //Calculate minimal value of data
```

```
  double min = List[0];
  for(int j=1; j < List.length; j++)
   if (List[j] < min)  min = List[j];
  //Calculate maximal value of data
  double max = List[0];
  for(int j=1; j < List.length; j++)
    if (List[j] > max)  max = List[j];
  System.out.println( "\nMax value = " + max);
  //Calculates number of classes
  int nClasses  = (int) ((max- infLimit)/intervalLength)+1;
  System.out.println( "Number of classes = " + nClasses);
  //Calculates interval borders and class markers
  //Grouping mode 2: defines inferior limit and length
  //of intervals.
  double classMarkers[] = new double[nClasses];
  double  Borders[] = new double[nClasses+2];
  Borders[0] = infLimit;
  classMarkers[0] = infLimit + intervalLength/2;
  for(int j=1; j < nClasses; j++)
    classMarkers[j] = classMarkers[j-1] + intervalLength;
  for(int j=1; j <= nClasses+1; j++)
    Borders[j] = Borders[j-1] + intervalLength;
  System.out.println( "\nBorders are" );
  printVector(Borders,nClasses+1 );
  System.out.println( "\nClass markers are" );
  printVector(classMarkers,nClasses );
 double FreqTable[][] = new double[nClasses][2];
 System.out.println(
 "\nEXTENDED TABLE OF ABSOLUTE FREQUENCIES" );
 //reuse
 FreqTable = group(List, Borders, classMarkers );
 meanVarFreqTable(FreqTable);
}


public static void groupM1()
{
//Declaration of data:
//observed values as a list
```

```
//Example 24 pag 12
double    List[] =
{
48.5, 49.2, 51.0, 50.9, 48.7, 50.5, 49.5, 50.8, 50.0, 51.1,
50.7, 51.1, 49.3, 49.1, 50.0, 48.7, 51.2, 49.2, 49.0, 49.3,
49.1, 48.7, 48.6, 49.2, 49.9, 50.1, 50.1, 50.9, 52.4, 50.2,
49.5, 50.9, 51.4, 49.7, 49.8, 50.8, 50.3, 51.8, 50.0, 51.1,
51.3, 50.1, 50.4, 51.1, 49.8, 49.8, 50.1, 50.7, 50.1, 49.9,
49.7, 51.1, 49.7, 49.9, 49.8, 50.5, 49.7, 50.8, 50.7, 50.7,
50.8,  51.5, 51.0
};
 System.out.println("\n\n");
 System.out.println("FROM DATA LIST TO FREQUENCY TABLE:");
 printList(List, "Raw data");
 //Borders of the intervals
 double  Borders[] = {48.5, 49.5, 50.5, 51.5, 52.5};
 double classMarkers[] = {49, 50, 51, 52};
 int nClasses = classMarkers.length;
 double FreqTable[][] = new double[nClasses][2];
 printList(Borders, "Borders of intervals");
 printList(classMarkers, "Class markers");
 FreqTable = group(List, Borders, classMarkers );
 meanVarFreqTable(FreqTable);
}



public static void groupM2( )
{
 //Declaration of data:
 //observed values as a list
 //Example 24 pag 12
 double    List[] =
{
48.5, 49.2, 51.0, 50.9, 48.7, 50.5, 49.5, 50.8, 50.0, 51.1,
50.7, 51.1, 49.3, 49.1, 50.0, 48.7, 51.2, 49.2, 49.0, 49.3,
49.1, 48.7, 48.6, 49.2, 49.9, 50.1, 50.1, 50.9, 52.4, 50.2,
49.5, 50.9, 51.4, 49.7, 49.8, 50.8, 50.3, 51.8, 50.0, 51.1,
51.3, 50.1, 50.4, 51.1, 49.8, 49.8, 50.1, 50.7, 50.1, 49.9,
```

```
49.7, 51.1, 49.7, 49.9, 49.8, 50.5, 49.7, 50.8, 50.7, 50.7,
50.8,  51.5, 51.0
   };
 System.out.println("\n\n");
 System.out.println("FROM DATA LIST TO FREQUENCY TABLE:");
 printList(List, "Raw data");

 //Calculate minimal value of data
 double min = List[0];
 for(int j=1; j < List.length; j++)
    if (List[j] < min)  min = List[j];
 System.out.println( "\nMin value = " + min);


 double infLimit = min;
 //Interval length
 double intervalLength = 1;
 System.out.println("Inferior limit = " + infLimit);
 System.out.println("Interval length = "
  + intervalLength);
 group(List, infLimit, intervalLength);
}

//Data is grouped and processed
public static void groupData( )
{
 //Method 1: all borders and class markers
 // are given by hand.
 groupM1();
 //Method 1: inferior border of first interval and
 //the length  of intervals are given.
 groupM2();
 separator();
}




//****************************
```

```
//Pstricks code
//**************************


//Produces the pstricks  code for the bar graph
//of a frequency table
public static void makeCode(double FreqTable[][])
{
 System.out.println("\nPSTRICKS code for the FreqTable\n");
 int n = FreqTable.length;
 //Width of each column
 double deltaX = 4;
 //The minimum x-coordinate is found
 double Minxx = 1000;
 for ( int i = 0; i < n; i++ )
  if (FreqTable[i][0] < Minxx)
        Minxx =   FreqTable[i][0];
 int Minxxx = (int) (deltaX*Minxx - deltaX);
 //The maximum x-coordinate is found
 double Maxx = 0;
 for ( int i = 0; i < n; i++ )
  if ( FreqTable[i][0] > Maxx)
      Maxx =    FreqTable[i][0];
 int Maxxx = (int) (deltaX*Maxx + deltaX);
 //The maximum y-coordinate is found
 int Maxyy = 0;
 for ( int i = 0; i < n; i++ )
  if (FreqTable[i][1] > Maxyy)
       Maxyy = (int) FreqTable[i][1] + 2;
 //Starting clause
 //x- and y-units.
 //The needed space is defined
 System.out.println("\\begin{center}" +
     "\n\\psset{xunit=0.15,yunit=0.3}" +
 "\n\\begin{pspicture}(" + Minxxx +",0)(" +
 Maxxx+ "," + Maxyy+")");
 //A rectangle is drawn for each entry in the
 //frequency table
```

```
 int lim1,lim2;
 for ( int i = 0; i < n; i++ )
  {
   lim1 = (int) (deltaX*FreqTable[i][0] -deltaX/2);
   lim2 = (int) (lim1 + deltaX);
   System.out.print(
 "\n\\psline("+ lim1 + ",0)" +
     "("+lim1 +"," + FreqTable[i][1]+ ")" +
     "("+lim2 + ","+ FreqTable[i][1]+ ")" +
     "("+lim2 + ","+ "0)"
    );
  }

 //Horizontal axis
 System.out.println("\n\\psline(" +
      Minxxx +",0)("+ Maxxx + ",0)");

 //Class markers (first coordinates) are indicated
 for ( int i = 0; i < n; i++ )
   {
int xposition = (int) (deltaX* FreqTable[i][0]);
int yposition = - 1;
    System.out.print
 (
  "\n\\rput*("+ xposition + "," + yposition+ "){"+
   (int) (FreqTable[i][0])+ "}"
 );
   }
 System.out.println();

 //Labels for heights (second coordinates) are indicated
 for ( int i = 0; i < n; i++ )
  {
   int xposition = (int) (deltaX* FreqTable[i][0]);
   int yposition = (int) (1+ FreqTable[i][1]);
   System.out.print
(
  "\n\\rput*("+ xposition + "," + yposition+ "){" +
      (int) (FreqTable[i][1])+ "}"
```

```
 );
 }

 //End clause
 System.out.println( "\n\\end{pspicture}");
 System.out.println( "\\end{center}");
 }

public static void codePSTricks()
 {
  System.out.println("\n\nPSTRICKS CODE FOR A BAR DIAGRAM");
  double  FreqT[][] =
        {{3, 3},
         {4, 3},
         {5, 2},
         {6, 2},
         {7, 1},
         {8, 1}};
 printMatrix(FreqT, "\n\nFREQUENCY  TABLE");
 makeCode(FreqT);
 separator();
 }




//================================================
//        BINOMIAL DISTRIBUTION
//================================================

//The  binomial distribution
//for  n repeats
//probability of head p,
//amplification factor F.
//Arithmetic trickery in action.
private static double[]  binomialDistribution(int n,
         double p, double F  )
{
 double pBinomial[] = new double[n+1];
```

```
 double pBinomialF[] = new double[n+1];
 double CoefficientsBinomial[] = new double[n+1];
 double biCoeff = 1;
 CoefficientsBinomial[0] = biCoeff;
 for(int r = 1; r < n+1; r++)
  {
   biCoeff = biCoeff * (n+1-r) / r;
   CoefficientsBinomial[r] = biCoeff;
  }
 System.out.println("\nCoefficients ");
 printVector(CoefficientsBinomial,n+1);
 System.out.println("Sum = " +
     sumVector(CoefficientsBinomial,n+1));

 for(int r = 0; r < n+1; r++)
  {
   pBinomial[r] = CoefficientsBinomial[r]  *
   Math.exp( r* Math.log(p) + (n-r) * Math.log(1-p));
  }
 System.out.println("\nProbabilities for 0,1,2...");
 System.out.println("Paste this vector to Excel" +
           " and make a graphic");
 printVector(pBinomial,n+1);
 sumVector(pBinomial,n+1);
 System.out.println("Sum = "
            +  sumVector(pBinomial,n+1));
 for(int k = 0; k < n+1; k++)
 pBinomialF[k] =  F* pBinomial[k];
 System.out.println("\nAbsolute frequencies for F = " + F);
 printVector(pBinomialF,n+1);
 System.out.println("Sum = "  +
           sumVector(pBinomialF,n+1));
 return pBinomial;
}

//Probability of closed region (extremes included)
private static double pClosedRegion( int limitInf,
                              int limitMax,
  double[] pBinomial)
```

```
{
 double  sum =0;
 for(int r = limitInf; r <= limitMax; r++)
  sum =  sum +  pBinomial[r];
 return sum;
}

//Detailed computation of the mean
private static void detailedMean(int n,
  double p,
  double[] pBinomial)
{
 System.out.println("\nDetailed mean: xp(x)");
 double  sum =0;
 double term;
 for(int r = 0; r < n+1; r++)
 {
  term =  r *  pBinomial[r];
  System.out.println( r + "    " + term) ;
  sum = sum +term;
 }
 System.out.println( "mean = sum xp(x) = " + sum) ;
 System.out.println( "Expected mean = np =  " + n*p) ;
}

//Detailed computation of the variance
private static void detailedVariance(int n,
  double p,
  double[] pBinomial)
{
 System.out.println("\nDetailed variance: " +
  "(x-mean)*(-mean)*p(x)");
 double  sum =0;
 double term;
 for(int r = 0; r < n+1; r++)
 {
  term =  Math.pow(r- n*p,2) *  pBinomial[r];
  System.out.println( r + "    " +  term) ;
  sum = sum +term;
```

```
 }
 System.out.println( "Variance = " +
"sum (x-mean)*(-mean)*p(x) = " + sum) ;
 System.out.println(
"Expected var = np(1-p) =  " + n*p*(1-p)) ;
 System.out.println( "Deviation = " + Math.sqrt(sum));
}


//Critical value for significance level alpha, one tail
private static void  critical1(  double[] pBinomial,
                                 double alpha )
{
 int numbTails = 1;
 System.out.println("\nLower tail");
 double cumulative = 0;
 double target = alpha;
 int i=-1;
 while( cumulative < target)
 {
  i = i+1;
  cumulative = cumulative + pBinomial[i];
  System.out.println(i + "  cumulative = " + cumulative);
 }
 System.out.println("To cumulate in the lower tail "
    + target
+ ", \nwe need the first " + i  + " Terms.");
 System.out.println("The critical value for alpha =  "
   + alpha
   + ", numb of tails = " + numbTails
   + "\nin the lower tail is "
   + i+ ". All terms below it are extreme.\n");
 System.out.println("Upper tail");
 int n = pBinomial.length;
 i= n;
 cumulative = 0;
 while( cumulative < target)
  {
   i = i-1;
```

```
   cumulative = cumulative + pBinomial[i];
   System.out.println(i + "  cumulative = " + cumulative);
  }

 System.out.println("To cumulate in the upper tail "
    + target
+ ", \nwe need term  " + i  + " and above.");
 System.out.println("The critical value for alpha =  "
   + alpha
   + ", numb of tails = " + numbTails
   + "\nin the upper tail is "
   + i + ". All terms above it are extreme.\n");
}


//Critical value for significance level alpha, two tails
private static  void  critical2(double[] pBinomial,
                            double alpha)
{
 critical1(pBinomial,alpha/2);
}


//Main of binomial distribution
private static void binomialDist(int n, double p, int F)
{
 double pBinomial[] = new double[n+1];
 System.out.println("n= " + n);
 System.out.println("p = " + p);
 System.out.println("F = " + F);
 pBinomial = binomialDistribution(n,p, F);
 detailedMean(n,p,pBinomial);
 detailedVariance(n,p,pBinomial);
 //Probability of a closed region (extremes included)
 int limitInf = 2;
 int limitMax = 4;
 double pR = pClosedRegion(limitInf,limitMax,pBinomial);
 System.out.println( "\nProb of closed region in " +
"within "+ limitInf + "  and " + limitMax +
```

```
 ",\nextremes included = " + pR) ;

}

//Critical values for a given
//significance level
private static void criticalValues(int n, double p,
                            double alpha)
{
 double pBinomial[] = new double[n+1];
 pBinomial = binomialDistribution(n,p, 1);
 System.out.println("Binomial distribution: \n" +
" n = " + n + ", p = " + p + ", " +
"significance level = alpha = " + alpha);
 System.out.println("\nCRITICAL VALUES");
 System.out.println("\nONE TAIL, alpha = " +alpha);
 System.out.println("\nTerms and cumulative values");
 critical1(pBinomial,alpha);
 //critical1(n,p, alpha);
 System.out.println("\nTWO TAILS, alpha = " +alpha);
 System.out.println("\nTerms and cumulative values");
 critical2(pBinomial, alpha);
 //critical2(n,p, alpha);
}

//Binomial distribution
public static void binomialDistribution()
{
 System.out.println("\n\nBINOMIAL DISTRIBUTION ");
 //Example 44 pag 32.
 int n= 6;
 double p = 0.25;
 //int F = 35;
 //binomialDist(n,p,F);
 //Inverse problem: find critical values
 double alpha = 0.02;
 criticalValues(n,p,alpha);
 separator();
}
```

```
//=================================================
// THE TOP 12  TESTS
//=================================================



//====== Methods of general use=========


//Returns numb!
private static long factorial (long numb)
{
 if (numb <=1 ) return 1;
else
 return numb * factorial (numb -1);
}

//Returns the double factorial
private static double doubleFact(double x)
{
 if (x <=0 ) return 1;
else
 return x * doubleFact(x -2);
}


//Returns the Gamma function
private static double Gamma(double x)
{
 int xRounded = (int) x;
 //System.out.println( x + "   " + xRounded);
 if (xRounded == x)
return factorial (xRounded-1);
 else
 {
 return Math.sqrt(pi)* doubleFact(2*xRounded -1) /
```

```
          Math.pow(2,xRounded);
 }
}



//======Inner top class that calculates==========
//===========critical and p-values ==============

//The methods that eventually can be useful
//to all distributions come here such as they are.
//The methods that are privative of each distribution
//are defined here as abstract
//and implemented in  specific classes below.
abstract class ValueAbstract
{
double sumzero;
double Error;
double jCritic;




//The density function is of abstract type.
//It is not implemented,
//so we can implement other functions too and
//process them with this very same machinery.
abstract double f(double t);



//========= p-values==========

//Returns the integral under the function f
//in within 0 and z. Precision=1/N.
//Simpson's rule adapted for reuse.
double HalfBodySimpson(double z, long N)
{
 double h = z/(2*N);
```

```
 long m = N;
 double sum = f(0);
 for(int j= 1; j<=m; j++)
 sum = sum +   4* f((2*j-1) * h);
 for(int j= 1; j<m; j++)
 sum = sum +   2* f((2*j) * h);
 sum = sum + f((2*m) * h);
 sum = (h/3)*sum;
 return sum;
}


//The area in within zero and z of chosen
//distribution is calculated
//for various degrees of precision.
double area(double z)
{
 z = (Math.sqrt(1+4*z*z)-1)/(2*z);
 int N;
 double  answer;
 int i = 3;
 N = (int) Math.pow(10, i);
 answer = HalfBodySimpson(z,N);
 return answer;
}

//Returns Psi(e) = p( x < e)
//the cumulative distribution function of e.
//This  depends
//on the distribution, it might be
//overridden in the corresponding classes.
double psi(double e)
{
 return 0.5 + area(e);
}

//Returns the p-value(e) = p( x > e)
//Local implementation of the p-value of  e
double pValue(double e)
```

```
{
 return 1- psi(e);
}


//Local manager
void pExecutor(double testValue)
{
 System.out.println("\nTest value = " + testValue);
 System.out.print("Simpson’s rule."+
  "\nArea = p( 0 <  x < " + testValue + ") = ");
 double area =  area(testValue);
 System.out.println( area);
 //psi(e) = p( x < e)
 double psi = psi(testValue);
 System.out.println("psi(" + testValue + ") " +
  "= p( x < " + testValue + ") = " +  psi );
 double pValue = pValue(testValue);
 System.out.println("p-value(" +
      testValue + ") = " +
    "p( x > " + testValue + ") = " + pValue);
}

//========Critical values=====

//Reports the transformed upper u-critical for
//the given distribution.
//Significance level = alpha.
//Precision=1/N.
//Simpson’s rule adapted for reuse.
//The error of approximation is also calculated.
double upperU(double h, double target)
{
 double r = h/6;
 jCritic = 0;
 double sum = sumzero;
 for(int j= 0; sum < target; j++)
  {
sum = sum +
```

```
    r*(f( j*h) + 4* f((j+0.5) * h) +  f((j+1) * h));
jCritic = j+1;
   }
 jCritic = jCritic-1;
 double greatSum = sum -
    r*(f( jCritic*h) + 4* f((jCritic+0.5) * h)
         +  f((jCritic+1) * h));
 Error = target-greatSum;
 double uAprox = (jCritic) *h;
 return uAprox;
}



//Returns the transformed value z = u/(1-u*u)
double z(double u)
{
 double z = u/(1-u*u);
 return z;
}

//The wide of subintervals is defined
double h()
{
 double u = 1;
 int i=3;
 long N = (int) Math.pow(10, i);
 double h = u/N;
 return h;
}

//Reports the upper z-critical for
//Significance level = alpha.
double vCrit(double target)
{
 double h = h();
 //System.out.println( "h = " + h);
 double uCritic =  upperU(h, target);
 double zCritic = z(uCritic);
```

```
 return zCritic;
}


//*****Analytical hunting*******

//Returns the density function of the distribution
abstract double ff(double z);



//Returns G(a,b) = h(f(a)  + 4 * ff((a+b)/2)
//                  + ff(b))-Error;
double G(double a, double b)
{
 double h = (b-a)/6;
 double  Gb = h* (  ff(a) + 4 * ff((a+b)/2)
                  + ff(b)  )-Error;
 return Gb;
}


//Returns the  ratio  G(zCritic,gi)/ ff(gi)
double ratio(double zCritic, double gi)
{
 double  ratio =  G(zCritic,gi)/ ff(gi) ;
 return ratio;
}

//Critical value for the upper tail.
//A professional  approximation is found by
//analytical hunting
double hunting(double zCritic )
{
 double h = h();
 double ur =  (jCritic+1)*h;
 //Right border of the critical interval
 double rBorder = z(ur);
 double gSubi = rBorder;
```

```
 double gSubiPlus= 0;
 int iterations = 0;
 int maxIterations = 30;
 boolean more = true;
 while (more)
  {
gSubiPlus = gSubi -  ratio(zCritic, gSubi);
iterations = iterations +1;
if (iterations >= maxIterations)  more = false;
if (gSubiPlus == gSubi) more = false;
gSubi = gSubiPlus;
  }
 return gSubiPlus;
}

//Critical value for significance level alpha, one tail
double  critical(  double alpha)
{
 double target = 1-alpha;
 double vCritic = vCrit(target);
 //vCritic = hunting(vCritic);
 return vCritic;
}

//Local manager
void critManager(double alpha)
{
System.out.println( "\nAlpha = Significance level = "
+ alpha + ".");

System.out.println("Critical values for ONE TAIL = "
+  critical(alpha) );

double alphaHalf = alpha/2;
System.out.println("Critical values for TWO TAILS = "
            + critical(alphaHalf) );
}

}//end of class ValueAbstract
```

```
//======Particular classes========


//The z class.
//Definition of the z density function
//and exclusive to it methods.
//It is an specification of the ValueAbstract class.
class zClass extends ValueAbstract
{

 //Constructor: this is the first thing to do
 zClass()
 {
  sumzero = 0.5; //half bell
  //System.out.println("\n===== z distribution");
}


//Function f is implemented here.
//Returns the transformed density function of the
//standardized normal distribution.
double f(double t)
{
 double z = t/(1-t*t);
 double  f = 1 /Math.pow(2*pi, 0.5)*Math.exp(-z*z/2)*
             (1+ t*t)/( (1-t*t)*(1-t*t));
 return f;
}


//Returns the density function of the Z distribution
double ff(double z)
{
 double  ff =  Math.exp(-z*z/2)/Math.pow(2*pi, 0.5);
 return ff;
}
```

```
}//end of class VzClass


//The t class
//Definition of the t density function
//and exclusive to it methods.
//It is an specification of the ValueAbstract class.
class tClass extends ValueAbstract
{

//Constructor: this is the first thing to do
tClass()
{
 sumzero = 0.5; //half bell
 //System.out.println("\n===== t distribution");
 //System.out.println("t degrees of freedom = " + dfT);
}



//Function f is implemented here.
//Returns the transformed density function of the
//t distribution with dfT d.f.
double f(double t)
{
 double nu = dfT;
 double z = t/(1-t*t);
 double  f = ( Gamma((nu+1)/2)
   / ( Math.sqrt(pi*nu)*Gamma(nu/2)) )
   * Math.pow(1+z*z/nu, -(nu+1)/2 )
                  * (1+ t*t)/( (1-t*t)*(1-t*t));
 return f;
}


//Returns the density function of the t-distribution
double ff(double z)
{
 double nu = dfT;
```

```
 double  ff =  ( Gamma((nu+1)/2)
    /  ( Math.sqrt(pi*nu)*Gamma(nu/2)) )
    *  Math.pow(1+z*z/nu, -(nu+1)/2 ) ;
 return ff;
}

}//end of class VtClass


//The chi2 class
//Definition of the Chi2 density function
//and exclusive to it methods.
//It is an specification of the ValueAbstract class.
class chi2Class extends ValueAbstract
{

//Constructor: this is the first thing to do
chi2Class()
{
 sumzero = 0; //no negative values
 //System.out.println("\n===== Chi2 distribution");
 /*System.out.println("Chi2 degrees of freedom = "
 + dfChi2);*/
}

//Returns Psi(e) = p( x < e)
//the cumulative distribution function of e
double psi(double e)
{
 return area(e);
}

//Function f is implemented here.
//Returns the transformed density function of the
//chi2 distribution with nu d.f.
double f(double t)
{
 double nu = dfChi2;
 double z = t/(1-t*t);
```

```
 double  f = ( 1 /
   ( Math.pow(2, nu/2)*Gamma(nu/2)) )
   * Math.exp(-z/2)* Math.pow(z, (nu-2)/2)
   * (1+ t*t)/( (1-t*t)*(1-t*t));
 return f;
}



//Returns the density function of the chi2-distribution
double ff(double z)
{
 double nu = dfChi2;
 double  ff = (1/(Math.pow(2, nu/2)*Gamma(nu/2)))
             * Math.exp(-z/2)* Math.pow(z, (nu-2)/2);
 return ff;
}


//Local manager
void critManager(double alpha)
{
 System.out.println("Critical values for ONE TAIL "
    + "\nof the chi-square- "
+  "distribution with df =  "
+ dfChi2
+ "\nAlpha = Significance level = "
+ alpha + ".");

 System.out.println("Lower critical value  = "
+  critical(1-alpha) );
 System.out.println("Upper critical value  = "
+  critical(alpha) );
 System.out.println("\nCritical values for TWO TAILS "
    + "\nof the chi-square "
+  "distribution with df=  "
+ dfChi2
+ "\nAlpha = Significance level = "
+ alpha + ".");
```

```
 double alphaHalf = alpha/2;
 double Chi2Menos = critical(1-alphaHalf) ;
 System.out.println("Lower critical value  = "
+  Chi2Menos  );
 double Chi2Mas = critical(alphaHalf) ;
 System.out.println("Upper critical value  = "
+  Chi2Mas  );
}
}//end of class pVChi2Class



//The F class
//Definition of the Chi2 density function
//and exclusive to it methods.
//It is an specification of the ValueAbstract class.
class FClass extends ValueAbstract
{

//Constructor: this is the first thing to do
FClass()
{
 sumzero = 0; //no negative values
}

//Returns a helping  function  with   d.f. =
//a(num) and b(den)
double g(double x, double a, double b)
{
double  g = Math.pow(x, a -1)*
                (Math.pow(1-x, b -1));
return g;
}


//Returns the integral under the function f
//in within 0 and u. Precision=1/N.
//Simpson's rule adapted for reuse.
//a and b are degrees of freedom
```

```
double HalfBodySimpsong(double u,
double a, double b, long N)
{
double h = u/(2*N);
long m = N;
double sum = g(0,a,b);
for(int j= 1; j<=m; j++)
 sum = sum +   4* g((2*j-1) * h,a,b);
for(int j= 1; j<m; j++)
 sum = sum +   2* g((2*j) * h,a,b);
sum = sum + g((2*m) * h,a,b);
sum = (h/3)*sum;
return sum;
}


//Returns the  beta  function in integral form
//with a and b d.f.
double beta( double a, double b)
{
double  answer;
int i = 4;
int  N = (int) Math.pow(10, i);
double u=1;
answer = HalfBodySimpsong(u,a,b,N);
return answer;
}


//****************F density function*************

//Returns the transformed density function of the
//F distribution with nu1 and nu2 d.f.
double FFunction(double t)
{
double nu1 = dfFNum;
double nu2 = dfFDen;
double z = t/(1-t*t);
double  f =   k*Math.pow(z, nu1/2-1)
```

```
                * Math.pow(nu2+nu1*z, -(nu1+nu2)/2)
            * (  (1+ t*t)/( (1-t*t)*(1-t*t))  ) ;
//System.out.println(f);
return f;
}


//To use this method for another distribution,
//instead of  FFunction(z), write and make a call for your
//new distribution.
double f(double z)
{
double  f = FFunction(z);
return f;
}


//Reports the transformed upper u-critical for
//the given distribution.
//Significance level = alpha.
//Precision=1/N.
//Simpson's rule adapted for reuse.
//The error of approximation is also calculated.
double upperU(double h, double target)
{
double nu1 = dfFNum;
double nu2 = dfFDen;
double a =  Math.pow(nu1, nu1/2);
double b =  Math.pow(nu2, nu2/2);
double c = beta(nu1/2,nu2/2);
k = a* b / c;
double r = h/6;
double sum = 0;
jCritic = 0;
for(int j= 0; sum < target; j++)
{
 sum = sum +
    r*(f( j*h) + 4* f((j+0.5) * h) +  f((j+1) * h));
jCritic = j+1;
```

```
}
jCritic = jCritic-1;
double greatSum = sum -
           r*(f( jCritic*h) + 4* f((jCritic+0.5) * h)
         +  f((jCritic+1) * h));
Error = target-greatSum;
double uAprox = (jCritic) *h;
return uAprox;
}

//Returns the transformed value z = u/(1-u*u)
double z(double u)
{
double z = u/(1-u*u);
return z;
}

//The wide of subintervals is defined
double h()
{
double u = 1;
int i=3;
long N = (int) Math.pow(10, i);
double h = u/N;
return h;
}

//Reports the upper z-critical for
//Significance level = alpha.
double vCrit(double target)
{
double h = h();
double uCritic =  upperU(h, target);
double zCritic = z(uCritic);
return zCritic;
}


//*****Analytical hunting*******
```

```
//Returns the density function of the t-distribution
double ff(double z)
{
double nu1 = dfFNum;
double nu2 = dfFDen;
double  f = k*Math.pow(z, nu1/2-1)
   * Math.pow(nu2+nu1*z, -(nu1+nu2)/2)   ;
return f;
}


//Returns F(a,b) = h(f(a)  + 4 * f((a+b)/2)  + f(b))-Error;
double F(double a, double b)
{
double h = (b-a)/6;

double  Fb = h* (  ff(a) + 4 * ff((a+b)/2)
                 + ff(b)  )-Error;
return Fb;
}


//Returns the  ratio  F(zCritic,gi)/ ff(gi)
double ratio(double zCritic, double gi)
{
double  ratio =  F(zCritic,gi)/ ff(gi) ;
return ratio;
}

//Critical value for the upper tail.
//A professional  approximation is found by
//analytical hunting
double hunting(double zCritic )
{
double h = h();
double ur =  (jCritic+1)*h;
//Right border of the critical interval
double rBorder = z(ur);
```

```
double gSubi = rBorder;
double gSubiPlus= 0;
int iterations = 0;
int maxIterations = 30;
boolean more = true;
while (more)
{
gSubiPlus = gSubi -  ratio(zCritic, gSubi);
iterations = iterations +1;
if (iterations >= maxIterations)  more = false;
if (gSubiPlus == gSubi) more = false;
gSubi = gSubiPlus;
}
return gSubiPlus;
}

//Critical value for significance level alpha, one tail
double  critical(  double alpha)
{
double target = 1-alpha;
double zCritic = vCrit(target);
zCritic = hunting(zCritic);
return zCritic;
}

//Local manager
void critManager(double alpha )
{
  System.out.println("\nF distribution");
System.out.println("Critical values for ONE TAIL "
    + "\nof the F- "
+ "distribution with df:  "
+ dfFNum + " and " + dfFDen+ ", "
+ "\nAlpha = Significance level = "
+ alpha + ".");

System.out.println("Lower critical value  = "
+  critical(1-alpha) );
System.out.println("Upper critical value  = "
```

```
+  critical(alpha) );

System.out.println("\nCritical values for TWO TAILS "
    + "\nof the f  "
+ "distribution."
+ "\nAlpha = Significance level = "
+ alpha + ".");
double alphaHalf = alpha/2;

System.out.println("Lower critical value  = "
+  critical(1-alphaHalf) );
System.out.println("Upper critical value  = "
+  critical(alphaHalf) );
}
}//end of class pVFClass



//========== Normal partitions ==========

//Let Z have a  standard normal distribution with
//given mean = 0  and deviation = 1.
//Which is the probability of the regions of
//the Z axis defined by a partition?
//Amplifying factor = F.

static void partitionZ(double[] partitionZ, int F)
{
 printList(partitionZ,"partitionZ");
 System.out.println("AmpliFactor = F = " + F);
 int l = partitionZ.length;
 double  Psi[] = new double[l+1];
 //Top abstract class is instructed to take
 //the form of the particular class
 p = z;
 //PSi values
 for(int i = 0; i< l; i++)
 Psi[i] =  p.psi(partitionZ[i]);
 Psi[l] = 1;
```

```
 printList(Psi,"Psi(partitionZ[i])");
 //Probabilities
 double prob[] = new double[l+1];
 prob[0] = Psi[0];
 for(int i = 1; i<= l; i++)
 prob[i] = Psi[i] -Psi[i-1];
 printList(prob,"Probabilities: prob[i] " +
          "= Psi[i] -Psi[i-1]");
 System.out.println("sum of probabilities = " +
 sumVector(prob, l+1));
 //Amplified prob
 double n[] = new double[l+1];
 for(int i = 0; i<= l; i++)
 n[i] = F*prob[i];
 printList(n,"Amplified probabilities = F*prob");
 System.out.println("sum of amplified probabilities = " +
 sumVector(n, l+1));
}

static void partitionZ()
{
 //Problem 8, pag. 102
 int F = 4500;
 double  partitionZ[] = {-1.2, 1.2, 2.2};
 partitionZ(partitionZ,F);
}

//Let X have a normal distribution with
//given mean = mu  and deviation = sigma.
//Which is the probability of the regions of
//the X axis defined by a partition?
//Amplifying factor = F.

static void partitionX()
 {
  //Example 79, pag. 57
  double mu =40;
  double sigma = 5;
  int F = 2500;
```

```
  double  partitionX[] = {32,45,53};
  printList(partitionX,"partitionX");
  System.out.println("Mean = mu = " + mu);
  System.out.println("Deviation = sigma = " +  sigma);
  System.out.println("Amplification factor = F = " + F);
  int l = partitionX.length;
  System.out.println("z-transform =  (z-mu)/sigma");
  double partitionZ[] = new double[l];
  for(int i = 0; i < l; i++)
  partitionZ[i] = (partitionX[i] - mu)/sigma;
  partitionZ(partitionZ,F);
 }


//=======================================
//======== THE 12 MOST USUAL TESTS ======
//=======================================


//======= 1: z-test for one mean ========

//Runs a z-test for the mean
//knowing the mean of a random sample
//and the populational variance.
//Confidence interval for the mean is also calculated.




static void printData1(double xBar, double mu,
double sigma, double n,  double alpha)
{
 System.out.println("\nz Test to study the null" +
 "\nhypothesis about a mean  given the mean " +
 "\nof a random sample and the standard deviation" +
 "\nof the population.\n");
 System.out.println("Observed mean = xBar = " + xBar);
 System.out.println("Ho: expected mean = mu = " + mu);
```

```
 System.out.println("sigma = " + sigma);
 System.out.println("n = " + n);
 System.out.println("alpha = " + alpha);
}


/*Measures the discrepancy between observed and expected
 * in  z Test to study the null hypothesis about a mean
* given the mean of a sample and the standard deviation
* of the population.*/
static double  zExpOneMean(double xBar, double mu,
double sigma, double n)
{
 System.out.println(
  "zExp = (xBar- mu)/( sigma/Math.sqrt(n))");
 double zExp = (xBar- mu)/( sigma/Math.sqrt(n));
 return zExp;
}

/*Confidence interval for the mean
given the populational variance */
static void confIntMeanZ(double xBar, double mu,
double sigma, double n,  double alpha)
{
 System.out.println( "\nConfidence interval of the mean" +
" \ngiven the populational deviation sigma");
 double alphaHalf = alpha/2;
 System.out.println( "alpha = " + alpha );
 double zc2 =  p.critical(alphaHalf);
 double r = zc2*sigma / Math.sqrt(n);
 System.out.println( "r = zc2*sigma / Math.sqrt(n)" );
 System.out.printf( "r = %10.4f \n", r );
 double limInf = xBar - r;
 double limSup = xBar + r;
 System.out.printf( "limInf = xBar - r " +
                      "= %10.4f \n" , limInf );
 System.out.printf( "limSup = xBar + r " +
                      "= %10.4f \n", limSup );
}
```

```
static //Standard error of the mean
double sError(double sigma, double n)
{
 System.out.println("\nStandard error  = " +
"sigma/Math.sqrt(n)");
 return sigma/Math.sqrt(n);
}

static //Needed n = size of the sample
//given tolerance E in the evaluation of the mean.
int nNeeded(double sigma, double E)
{
 System.out.println("\nTolerance error  E = "+ E);
 //Critical z with two tails
 double alphaHalf = alpha/2;
 double zc2 =  p.critical(alphaHalf);
 System.out.println("m= " + "zc2 * sigma / E ");
 //Needed n
 double m = zc2 * sigma / E;
 System.out.println("Needed  n = m*m");
 //Upwards approximation
 return (int) Math.ceil(m*m);
}


//Runs a z test for one mean
public static void zTestOneMeanWork(double xBar, double mu,
double sigma, double n,  double alpha )
{

//Top class is instructed to take
//the form of the particular class
p = z;
//z-test for one mean
printData1(xBar,mu,sigma,n,alpha);
//Critical values
double zCriticOneTail = p.critical(alpha);
double alphaHalf = alpha/2;
```

```
double zCriticTwoTails = p.critical(alphaHalf);
//z-test
System.out.printf( "Discrepancy between observed and ");
System.out.printf( "expected: \n");
double zExp = zExpOneMean(xBar,mu,sigma,n);
System.out.printf( "zExp = %10.4f \n" , zExp);
double pValue = p.pValue(zExp);
System.out.println("p-value(" +
      zExp + ") = " +
    "p( x > " + zExp + ") = " + pValue);
System.out.printf("zCritic for TWO TAILS " +
"= zc2 =  +/-%10.4f \n",  zCriticTwoTails);
System.out.printf("zCritic for ONE TAIL  " +
"= zc1 = +/-%10.4f \n",  zCriticOneTail);
//Standard error
double error = sError(sigma,n);
System.out.printf("Standard error  = %10.4f \n", error);
//Confidence interval
confIntMeanZ(xBar,mu,sigma,n,alpha);
//Admitted tolerance for mu
double E = 5;
//Size of sample to attain tolerance E
int nNeeded = nNeeded(sigma,  E);
System.out.printf("Needed n = %10d \n", nNeeded);
separator();
}



//Data input for  a z test for one mean
public static void zTestOneMean( )
{
//Example 184, pg 169
double xBar = 40;
double mu = 20;
double sigma = 16;
double n = 36;
alpha = 0.05;
zTestOneMeanWork(xBar,mu,sigma,n,alpha);
```

```
}

//======= 2: z-test for two means ========



static void printData2(double xBar, double yBar,
               double mux, double muy,
            double sigmax, double sigmay,
            double nx,  double ny,
            double alpha)
{
System.out.println("\nz Test to study the null" +
"\nhypothesis about the difference between two means" +
"\ngiven the means of two random samples and " +
"\nthe standard deviations of the populations\n");
 System.out.println("Observed means: ");
 System.out.println("xBar = " + xBar);
 System.out.println("yBar = " + yBar);
 System.out.println("Ho: expected means: ");
 System.out.println("mux = " + mux);
 System.out.println("muy = " + muy);
 System.out.println("sigmax = " + sigmax);
 System.out.println("sigmay = " + sigmay);
 System.out.println("nx = " + nx);
 System.out.println("ny = " + ny);
 System.out.println("alpha = " + alpha);

}

/*Measures the discrepancy between observed and expected
 * in  z Test to study the null hypothesis about
 * the difference between tow means.*/
static double  zExpTwoMeans(double xBar, double yBar,
        double mux, double muy,
        double sigmax, double sigmay,
        double nx,  double ny)
{
 double jointDev =
```

```
 Math.sqrt(sigmax*sigmax/nx + sigmay*sigmay/ny);
 System.out.println( "jointDev = " +
 " Math.sqrt(sigmax*sigmax/nx + sigmay*sigmay/ny);" );
 System.out.println( "jointDev = " + jointDev);
 System.out.printf( "Discrepancy between observed and ");
 System.out.printf( "expected: \n");
 System.out.println(
       "zExp = ( (xBar-yBar) -( mux-muy)) / jointDev " );
 double zExp = ( (xBar-yBar) -( mux-muy))/jointDev;
 return zExp;
}

/*Confidence interval for the difference of means
given populational variances */
static void confIntDiffMeansZ(double xBar, double yBar,
        double sigmax, double sigmay,
        double nx,  double ny,
        double alpha)
{
 System.out.println( "\nConfidence interval of the " +
" \ndifference between means " +
"given the populational deviations");
 double alphaHalf = alpha/2;
 System.out.println( "alpha = " + alpha );
 double zc2 =  p.critical(alphaHalf);
System.out.println("Critical z  =  zc2 = +/-"
            + zc2);
 double jointDev =
 Math.sqrt(sigmax*sigmax/nx + sigmay*sigmay/ny);
 System.out.println( "jointDev = " +
  " Math.sqrt(sigmax*sigmax/nx + sigmay*sigmay/ny);" );
 System.out.println( "jointDev = " + jointDev );
 double r =  zc2 * jointDev;
 System.out.println( "r = zc2 * jointDev" );
 System.out.printf( "r = %10.4f \n", r );
double limInf2 =  xBar-yBar - r;
double limSup2 =  xBar-yBar + r;
 System.out.printf( "limInf = xBar-yBar - r " +
                     "= %10.4f \n" , limInf2 );
```

```
 System.out.printf( "limSup = xBar-yBar + r " +
                      "= %10.4f \n", limSup2 );

}

/*z Test to study the null hypothesis about the difference
 *between two means
 * given the means of   samples and  standard deviations
 * of the population.*/
private static void  zTestTwoMeansWork(
double xBar, double yBar,
        double mux, double muy,
        double sigmax, double sigmay,
        double nx,  double ny,
        double alpha)
{
 //Top abstract class is instructed to take
 //the form of the particular class
 p = z;
 printData2(xBar, yBar,mux, muy,
           sigmax,  sigmay,
           nx,   ny,
           alpha);
 //Critical values
 double zCriticOneTail = p.critical(alpha);
 double alphaHalf = alpha/2;
 double zCriticTwoTails = p.critical(alphaHalf);
 //z-test
 double zExp = zExpTwoMeans(xBar, yBar,mux, muy,
                            sigmax,  sigmay,
                            nx,   ny );
 System.out.printf( "zExp = %10.4f \n" , zExp);
 //p-value
 double pValue = p.pValue(zExp);
 System.out.println("p-value(" +     zExp + ") = " +
      "p( x > " + zExp + ") \n = " + pValue);
 System.out.printf("zCritic for TWO TAILS " +
  "= zc2 =  +/-%10.4f \n",  zCriticTwoTails);
 System.out.printf("zCritic for ONE TAIL  " +
```

```
   "= zc1 = +/-%10.4f \n",  zCriticOneTail);
 //Confidence interval
 confIntDiffMeansZ(xBar, yBar,
           sigmax,  sigmay,
           nx,   ny,
           alpha);
 separator();
}

/*Data input to a z Test to study
 * the null hypothesis about the difference
 *between two means
 * given the means of   samples and  standard deviations
 * of the population.*/
private static void  zTestTwoMeans()
{
 //Example 190 pag. 172
 double mux = 3.2;
 double muy = 4.5;
 double sigmax = 1;
 double sigmay = Math.sqrt(0.2);
 double xBar = 4.2;
 double yBar = 3.8;
 double nx = 4;
 double ny = 4;
 double alpha = 0.05;
 zTestTwoMeansWork(xBar, yBar,mux, muy,
                   sigmax,  sigmay,
                   nx,   ny,
                   alpha);
}


//======= 3: t-test for one mean ========

static void printData3(double xBar, double mu,
double s, double n,  double alpha)
{
System.out.println("\nt-Test to study the null" +
```

```
 "\nhypothesis about a mean  given the mean and deviation"
 + "\nof a random sample.\n");
System.out.println("Observed mean = xBar = " + xBar);
System.out.println("Ho: expected mean = mu = " + mu);
System.out.println("deviation = " + s);
System.out.println("n = " + n);
System.out.println("alpha = " + alpha);
}

/*Measures the discrepancy between observed and expected
* in  a t-Test to study the null hypothesis about a mean
* given the mean and deviation of a sample.*/
static double  tExpOneMean(double xBar, double mu,
double s, double n)
{
System.out.println(
  "tExp = (xBar- mu)/( s/Math.sqrt(n))");
double tExp = (xBar- mu)/( s/Math.sqrt(n));
return tExp;
}


/*Confidence interval for the mean
given the populational variance */
static void confIntMeanT(double xBar, double mu,
double s, double n,  double alpha)
{
System.out.println( "\nConfidence interval of the mean" +
" \ngiven the sample deviation s");
double alphaHalf = alpha/2;
System.out.println( "alpha = " + alpha );
dfT = n-1;
System.out.println( "dfT = " + dfT);
double tc2 =  p.critical(alphaHalf);
double r = tc2*s / Math.sqrt(n);
System.out.println( "r = tc2*sigma / Math.sqrt(n)" );
System.out.printf( "r = %10.4f \n", r );
double limInf = xBar - r;
double limSup = xBar + r;
```

```
System.out.printf( "limInf = xBar - r " +
                        "= %10.4f \n" , limInf );
System.out.printf( "limSup = xBar + r " +
                        "= %10.4f \n", limSup );
}

/*t Test to study the null hypothesis about a mean
 * given the mean of a sample and the standard deviation
 * of the sample.*/
private static void tTestOneMeanWork(
double xBar, double mu,
double s, double n,  double alpha)
{
 //Degrees of freedom
 dfT = n-1;
 //Truncation of degrees of freedom
 if (dfT > 40) dfT =  40;
 //Top abstract class is instructed to take
 //the form of the particular class
 p = t;
 //z-test for one mean
 printData3(xBar,mu,s,n,alpha);
 System.out.println( "dfT = " + dfT);
 //Critical values
 double tCriticOneTail = p.critical(alpha);
 double alphaHalf = alpha/2;
 double tCriticTwoTails = p.critical(alphaHalf);
 //z-test
 System.out.printf( "Discrepancy between observed and ");
 System.out.printf( "expected: \n");
 double tExp = tExpOneMean(xBar,mu,s,n);
 System.out.printf( "tExp = %10.4f \n" , tExp);
 double pValue = p.pValue(tExp);
 System.out.println("p-value(" +
        tExp + ") = " +
      "p( x > " + tExp + ") \n = " + pValue);
 System.out.printf("t-Critic for TWO TAILS " +
  "= tc2 =  +/-%10.4f \n",  tCriticTwoTails);
 System.out.printf("zCritic for ONE TAIL  " +
```

```
  "= tc1 = +/-%10.4f \n",  tCriticOneTail);
 //Confidence interval
 confIntMeanT(xBar,mu,s,n,alpha);
 separator();
}

/* Data input for a t Test to study
 * the null hypothesis about a mean
 * given the mean of a sample and the standard deviation
 * of the sample.*/
private static void tTestOneMean()
{
 //Example 194 pag. 175
 double n = 20;
 double xBar = 7;
 double s = 1.3;
 double mu = 5;
 double alpha = 0.05;
 tTestOneMeanWork(xBar,mu,s,n,alpha);
}


//======= 4: z-test for one proportion ========

/* z Test to study the null hypothesis about
 * a proportion given the observed relative frequency
 *  of a random sample.
 */

static void printData4(double x, double n,
           double p,  double alpha)
{
 System.out.println("\nz Test to study the null" +
 "\nhypothesis about a proportion  given the  " +
 "\nobserved relative frequency of a random sample.\n");
 System.out.println("Observed events = x = " + x);
 System.out.println( "n = " + n);
 System.out.println("Observed relative frequency = f = " +
               "x/n ="  + x/n);
```

```
 System.out.println("Ho: expected proportion = p = " + p);
 System.out.println("n = " + n);
 System.out.println("alpha = " + alpha);
}


/*Measures the discrepancy between observed and expected
 * in a z-Test to study the null hypothesis about a
 * proportion given the observed relative frequency
 * of a random sample. */
static double  zExpOneProp(double x, double n,
 double p)
{
 double f = x/n;
 System.out.println(
  "zExp = (f- p)/( Math.sqrt( p*(1-p)/n))");
 double zExp = (f- p)/( Math.sqrt( p*(1-p)/n));
 return zExp;
}

/*Confidence interval for the proportion
given the observed relative frequency */
static void confIntOnePropZ(double x, double n,
                    double alpha)
{
 System.out.println( "\nConfidence interval " +
  "of the proportion" +
" \ngiven the observed relative frequency");
 double alphaHalf = alpha/2;
 System.out.println( "alpha = " + alpha );
 double zc2 =  p.critical(alphaHalf);
 double f = x/n;
 System.out.println( "f = x/n= " + f);
 double r = zc2 * Math.sqrt(f*(1-f)/n);
 System.out.println( "r = zc2 * Math.sqrt(f*(1-f)/n)" );
 System.out.printf( "r = %10.4f \n", r );
 double limInf = f - r;
 double limSup = f + r;
```

```
 System.out.printf( "limInf = xBar - r " +
                      "= %10.4f \n" , limInf );
 System.out.printf( "limSup = xBar + r " +
                      "= %10.4f \n", limSup );
}


//Needed n = size of the sample
//given tolerance E in the evaluation of the proportion.
static int nNeededOneProp(double alpha, double E)
{
 System.out.println("\nTolerance error  E = "+ E);
 //Critical z with two tails
 double alphaHalf = alpha/2;
 double zc2 =  p.critical(alphaHalf);
 System.out.println("m= " + "zc2 *  /( 2* E) ");
 //Needed n
 double m = zc2  /( 2* E) ;
 System.out.println("Needed  n = m*m");
 //Upwards approximation
 return (int) Math.ceil(m*m);
}


//Runs a z test for one proportion
public static void zTestOnepropWork(double x, double n,
        double pzero,  double alpha)
{
 //Top class is instructed to take
 //the form of the particular class
 p = z;
 //z-test for one mean
 printData4(x,n,pzero,alpha);
 //Critical values
 double zCriticOneTail = p.critical(alpha);
 double alphaHalf = alpha/2;
 double zCriticTwoTails = p.critical(alphaHalf);
 //z-test
 System.out.printf( "Discrepancy between observed and ");
```

```
 System.out.printf( "expected: \n");
 double zExp = zExpOneProp(x,n,pzero);
 System.out.printf( "zExp = %10.4f \n" , zExp);
 double pValue = p.pValue(zExp);
 System.out.println("p-value(" +
      zExp + ") = " +
    "p( x > " + zExp + ") = " + pValue);
 System.out.printf("zCritic for TWO TAILS " +
"= zc2 =  +/-%10.4f \n",  zCriticTwoTails);
 System.out.printf("zCritic for ONE TAIL  " +
"= zc1 = +/-%10.4f \n",  zCriticOneTail);
 //Confidence interval
 confIntOnePropZ(x, n,  alpha);
 //Admitted tolerance for p
 double E = 0.1;
 //Size of sample to attain tolerance E
 int nNeeded = nNeededOneProp(alpha,  E);
 System.out.printf("Needed n = %10d \n", nNeeded);
 separator();
}

//Data input for  a z-test for one proportion
private static void zTestOneProp()
 {
  //Example 196, pag. 177
  double x = 5;
  double n = 20;
  double p = 0.3;
  double alpha = 0.05;
  zTestOnepropWork(x, n,  p,  alpha);
 }



//======= 5: z-test for two proportions ========


/*z-Test for the comparison of two proportions
 * against the observed frequencies in two samples.
```

```
 */

static void printData5(double x, double m,
               double y, double n,
                       double p,  double r, double alpha)
{
 System.out.println("\nz Test to study the null" +
 "\nhypothesis about the difference between " +
 "two proportions  \ngiven the  " +
 "\nobserved relative frequencies" +
 " of two random samples.\n");
 System.out.println("Observed events :");
 System.out.println( "x = " + x);
 System.out.println( "m = " + m);
 System.out.println( "y = " + y);
 System.out.println( "n = " + n);
 double f = x/m;
 System.out.println( "f = = x/m = " + f);
 double g = y/n;
 System.out.println( "g = = y/n = " + g);
 System.out.println( "p = " + p);
 System.out.println( "r = " + r);
 double pMinusR = p-r;
 System.out.println("Ho: expected difference" +
              " between proportions: ");
 System.out.println( "p-r = " + pMinusR);
 double fMinusg = f-g;
 System.out.println("Observed difference" +
       " between relative frequencies:\nf-g=  "+ fMinusg);
 System.out.println("alpha = " + alpha);
 double sBi = Math.sqrt( p*(1-p)/m+ r*(1-r)/n);
 System.out.println("sBi = Math.sqrt( p*(1-p)/m" +
              "+ r*(1-r)/n) = " + sBi);
}


/*Measures the discrepancy between observed and expected
* in a z-Test to study the null hypothesis about
* the difference between two proportions given
```

```
* the observed relative frequencies
* of two random samples. */
static double  zExpTwoProp(double x, double m,
                           double y, double n,
                           double p,  double r)
{
 double f = x/m;
 double g = y/n;
 double num = (f-g)-( p-r);
 System.out.println( "num = (f-g)-( p-r) = " + num);
 //Binomial deviation
 double sBi = Math.sqrt( p*(1-p)/m+ r*(1-r)/n);
 System.out.println( "zExp = num/sBi");
 double zExp = num/sBi;
 return zExp;
}

/*Confidence interval for the difference  between two
 * proportions    given the observed relative frequencies */
static void confIntTwoPropZ(double x, double m,
        double y, double n, double alpha)
{
 System.out.println( "\nConfidence interval " +
       "of the difference between proportions" +
       " \ngiven the observed relative frequencies");
 double alphaHalf = alpha/2;
 System.out.println( "alpha = " + alpha );
 double zc2 =  p.critical(alphaHalf);
 double f = x/m;
 double g = y/n;
 double sBiHat = Math.sqrt( f*(1-f)/n+ g*(1-g)/n);
 System.out.println("sBiHat = " +
"Math.sqrt( f*(1-f)/n+ g*(1-g)/n)"
 + sBiHat);
 double rr = zc2 * sBiHat;
 System.out.println( "rr = zc2 * sBiHat" );
 System.out.printf( "rr = %10.4f \n", rr );
 double limInf = f -g- rr;
 double limSup = f -g + rr;
```

```
 System.out.printf( "limInf = xBar - rr " +
           "= %10.4f \n" , limInf );
 System.out.printf( "limSup = xBar + rr " +
           "= %10.4f \n", limSup );
}


static void more(double x, double m,
        double y, double n,
        double pzero,  double r, double alpha)
{
 double f = x/m;
 double g = y/n;
 double pHat = (x + y)/(m+n);
 System.out.println( "\npHat =  (x + y)/(m+n) = " + pHat);
 double sBiHat = Math.pow( pHat*(1-pHat)*(1/m+1/n), 0.5);
 System.out.println( "\nsBiHat = " +
  "Math.pow( pHat*(1-pHat)*(1/m+1/n), 0.5)");
 System.out.println( "sBiHat = " + sBiHat);
 double zExpHat = ((f-g)-(pzero-r))/sBiHat;
 System.out.println( "zExpHat = ((f-g)-(p-r))/sBiHat");
 System.out.println( "zExpHat = " + zExpHat);
 double sApprox = Math.pow( f*(1-f)/m + g*(1-g)/n, 0.5);
 System.out.println( "\nsApprox = " +
  "Math.pow( f*(1-f)/m + g*(1-g)/n, 0.5)" );
 System.out.println( "sApprox = " + sApprox);
 double zA = ( (f-g) - (pzero-r))/(sApprox);
 System.out.println( "zExp approx = " +
  "( (f-g) - (p-r))/(sApprox)" );
 System.out.println( "zEXp approx = " + zA);
 double alphaHalf = alpha/2;
 double zCritic = p.critical(alphaHalf);
 double b1 = f-g - zCritic* sBiHat;
 double b2 = f-g + zCritic* sBiHat;
 double c1 = f-g - zCritic* sApprox;
 double c2 = f-g + zCritic* sApprox;
 double fMinusg = f-g;
 System.out.println( "\nf-g = " +fMinusg);
 System.out.println( "Borders of IC = f-g +/- zCritic* s");
```

```
 System.out.println( "IC BiHAt (" + b1 +
 ", " + b2 + ")");
 System.out.println( "IC approx (" + c1 +
 ", " + c2 + ")");
}

//Runs a z test for one proportion
public static void zTestTwopropWork(double x, double m,
        double y, double n,
        double pzero,  double r, double alpha)
{
 //Top class is instructed to take
 //the form of the particular class
 p = z;
 //z-test for one mean
 printData5(x,m,y,n,pzero,r,alpha);
 //Critical values
 double zCriticOneTail = p.critical(alpha);
 double alphaHalf = alpha/2;
 double zCriticTwoTails = p.critical(alphaHalf);
 //z-test
 System.out.printf( "Discrepancy between observed and ");
 System.out.printf( "expected: \n");
 double zExp = zExpTwoProp(x,m,y,n,pzero,r);
 System.out.printf( "zExp = %10.4f \n" , zExp);
 double pValue = p.pValue(zExp);
 System.out.println("p-value(" +
 zExp + ") = " +
 "p( x > " + zExp + ") = " + pValue);
 System.out.printf("zCritic for TWO TAILS " +
 "= zc2 =  +/-%10.4f \n",  zCriticTwoTails);
 System.out.printf("zCritic for ONE TAIL  " +
 "= zc1 = +/-%10.4f \n",  zCriticOneTail);
 //Confidence interval
 confIntTwoPropZ(x,m,y,n,alpha);
 more(x,m,y,n,pzero,r,alpha);
 separator();
}
```

```
//Data input for a z-test for two proportions
public static void zTestTwoProp()
 {
  double x = 235;
  double m = 300;
  double y = 70;
  double n = 200;
  double p =  0.7;
  double r = 0.3;
  double alpha = 0.05;
  zTestTwopropWork(x,m,y,n,p,r,alpha );
 }

//======= 6: chi2-test for one  variance ========

//Runs a chi2-test for one variance.
//Confidence interval  is also calculated.

static void printData6(double sigma,
double s, double n,  double alpha)
{
System.out.println("\nchi2-Test to study the null" +
 "\nhypothesis about a variance  given the  deviation" +
 "\nof a random sample.\n");
System.out.println("Observed deviation = s = " + s);
System.out.println("Observed variance = sˆ2 = " + s*s);
System.out.println("Ho: populational variance = " +
"sigmaˆ2 = " + sigma*sigma);
System.out.println("n = " + n);
double df = n-1;
System.out.println("Degrees of freedom of the chi2 = "
+ df);
System.out.println("alpha = " + alpha);
}

/*Measures the discrepancy between observed and expected
* in  a t-Test to study the null hypothesis about a mean
```

```
* given the mean and deviation of a sample.*/
static double  chi2ExpOneVar(double sigma,
double s, double n)
{
 double var = s*s;
 double sigma2 = sigma*sigma;
 System.out.println(
  "chi2Exp = (n-1)* var/sigma2");
 double chi2Exp = (n-1)* var/sigma2;
return chi2Exp;
}


/*Confidence interval for the populational variance */
static void confIntOneVarChi2(double sigma,
double s, double n,  double alpha)
{
 System.out.println( "\nConfidence interval of the variance"
" \ngiven the sample variance s*s");
 double alphaHalf = alpha/2;
 System.out.println( "alpha = " + alpha );
 dfChi2 = n-1;
 System.out.println("Degrees of freedom of the chi2 = "
+ dfChi2);
 double Chi2Plus=  p.critical(alphaHalf);
 double Chi2Minus=  p.critical(1-alphaHalf);
 System.out.println("Upper critical chi2 value = " +
  "chi2Plus = "  + Chi2Plus);
 System.out.println("Lower critical chi2 value = " +
  "chi2Minus ="  + Chi2Minus);
 double limInf = s*s*(n-1)/(Chi2Plus);
 double limSup = s*s*(n-1)/(Chi2Minus);
 System.out.printf( "limInf = var*(n-1)/(Chi2Plus) " +
                      "= %10.4f \n" , limInf );
 System.out.printf( "limSup = var*(n-1)/(Chi2Minus" +
                      "= %10.4f \n", limSup );
 System.out.println("Confidence interval " +
 "for the variance \n  = (" + limInf + "," + limSup + ")");
 limInf = Math.sqrt(limInf);
```

```
 limSup =  Math.sqrt(limSup);
 System.out.println("Confidence interval " +
 "for the deviation \n  = (" + limInf + "," + limSup + ")");



}

/* chi2-Test to study
 * the null hypothesis about a variance
 * given the variance of a sample.*/

private static void chi2TestOneVarWork(double sigma,
double s, double n,  double alpha)
{
 //Degrees of freedom
 dfChi2 = n-1;
 //Top abstract class is instructed to take
 //the form of the particular class
 p = chi2;
 //z-test for one mean
 printData6(sigma,s,n,alpha);
 System.out.println("Degrees of freedom of the chi2 = "
+ dfChi2);
 //Critical values
 System.out.printf( "Discrepancy between observed and ");
 System.out.printf( "expected: \n");
 double chi2Exp = chi2ExpOneVar(sigma,s,n);
 System.out.printf( "chi2Exp = %10.4f \n" , chi2Exp);
 double pValue = p.pValue(chi2Exp);
 System.out.println("p-value(" +
        chi2Exp + ") = " +
      "p( x > " + chi2Exp + ") \n = " + pValue);
 p.critManager(alpha);
 //Confidence interval
 confIntOneVarChi2(sigma,s,n,alpha);
 separator();
}
```

```
/* Data input for a t Test to study
 * the null hypothesis about a mean
 * given the mean of a sample and the standard deviation
 * of the sample.*/
private static void chi2TestOneVar()
{
 //Examples 204 and 206, pag. 182
 double sigma = Math.sqrt(5);
 double s = 2;
 double n = 9;
 double alpha = 0.02;
 chi2TestOneVarWork(sigma,s,n,alpha);
}



//======= 7: F-test for two  variances ========

//Runs an F-test for variances.
//Confidence interval for the relation is also calculated.

static void printData7(double varx, double vary, double R,
 double nx, double ny,  double alpha)
{
 System.out.println("\nF-Test to study the null" +
 "\nhypothesis about the relation of two variances.\n");
 System.out.println("Observed data: " );
 System.out.println("\nvarx =" +  varx);
 System.out.println("nx = " + nx);
 System.out.println("vary = " + vary);
 System.out.println("ny= " + ny);

 System.out.println("Ho: expected relation" +
" between populational variances = \nR =" +R);
 System.out.println("alpha = " + alpha);
 dfFNum = nx-1;
 dfFDen = ny-1;
```

```
 System.out.println("Degrees of freedom of the numerator= "
+ dfFNum);
 System.out.println("Degrees of freedom of" +
  " the denumerator= " + dfFDen);
 System.out.println("alpha = " + alpha);
}

/*Measures the discrepancy between observed and expected
* in  an F-Test to study the null hypothesis about
* the relation between two variances.*/
static double  FExpTwoVar(double varx, double vary,
             double R, double nx, double ny)
{
 double RExp= varx/vary;
 dfFNum = nx-1;
 dfFDen = ny-1;
 System.out.println("\nRExp = varLarge/varSmall");
 System.out.println("RExp = " + RExp);
 double FExp = RExp/R;
 System.out.println("\nFExp = RExp/R");
 System.out.println("RExp = " + FExp);
 return FExp;
}


public static void ConfIntervalRelat2VarsF(double varx,
 double vary, double R,
 double nx, double ny,  double alpha)
{
 System.out.println("\nConfidence interval " +
  "for the relation between variances:");
 double alphaHalf = alpha/2;
 System.out.println( "alpha = " + alpha );
 dfFNum = nx -1;
 dfFDen = ny-1;
 System.out.println("Degrees of freedom of the numerator= "
+ dfFNum);
 System.out.println("Degrees of freedom of" +
  " the denumerator= " + dfFDen);
```

```
 double FMinus =  p.critical(1-alphaHalf);
 double FPlus =  p.critical(alphaHalf);
 System.out.println("FMinus = " + FMinus);
 System.out.println("FPlus = " + FPlus);
 double limInf = varx/(vary*FPlus);
 double limSup = varx/(vary*FMinus);
 System.out.println("limInf = varx/(vary*FPlus) = "
            + limInf);
 System.out.println("limSup = varx/(vary*FMinus) = "
            + limSup);
 System.out.printf("Confidence interval = " +
  "( %10.4f,  %10.4f)", limInf, limSup);
}

// F-test for the
//comparison of two variances.
private static void FTestTwoVariancesWork(
 double varx, double vary, double R,
 int nx, int ny,  double alpha )
{
 dfFNum = nx -1;
 dfFDen = ny-1;
 double nu1 = dfFNum;
 double nu2 = dfFDen;
 double a =  Math.pow(nu1, nu1/2);
 double b =  Math.pow(nu2, nu2/2);
 double c = F.beta(nu1/2,nu2/2);
 k = a* b / c;
 printData7(varx, vary, R,   nx, ny,  alpha);
 System.out.println("F degrees of freedom: "
             + "Num = " + dfFNum
             + ", Den = " + dfFDen);
 //F-test
 System.out.printf(
 "Discrepancy between observed and ");
 System.out.printf( "expected: \n");
 double FExp = FExpTwoVar(varx, vary, R,   nx, ny);
 System.out.printf( "FExp = %10.4f \n" , FExp);
 //Top class is instructed to take
```

```
 //the form of the particular class
 p = F;
 double pValue = p.pValue(FExp);
 System.out.println("p-value(" +
 FExp + ") = " +
 "p( x > " + FExp + ") = " + pValue);
 //Critical values
 p.critManager(alpha);
 //Confidence interval
 ConfIntervalRelat2VarsF(varx, vary, R,
                 nx, ny,  alpha);
 separator();
}

//Data input for an F-test for the
//comparison of two variances.
private static void FTestTwoVariances( )
 {
  //Example 211, pag. 187.
  //varx is the larger variance
  double varx =  3;
  int nx = 4;
  //vary is the smaller variance
  double vary = 0.25;
  int ny = 4;
  //R = Ratio of greater variance to the smaller
  double R = 1;
  double alpha = 0.02;
  FTestTwoVariancesWork(varx, vary, R,
             nx, ny,  alpha);
 }

//======= Bivariate tests about means ========

//To compare means, one establishes in first place
//whether or not populational variances are equal.
// If they are equal, means are compared
//following one procedure in which
//samples are joint together to estimate the unique
```

```
//variance, but if they are not, another procedure
//is used in which samples go apart.


//==== 8: comparing means when variances are equal ====

//t-test to compare the difference between means
//against an expected difference.
//Variances are supposed to be equal.
//Thus, samples are joint together to estimate
//the unique variance.

static void printData8(
 double meanx,
 double varx,
 double nx,
 double meany,
 double vary,
 double ny,
 double mux,
 double muy,
 double alpha,
 String s)
{
 System.out.println("\nt-Test to study the null" +
 "\nhypothesis about the differece between" +
 " two means\n" + s + "\n");
 System.out.println("Observed data: " );
 System.out.println("\nmeanx=" +  meanx);
 System.out.println("varx =" +  varx);
 System.out.println("nx = " + nx);
 System.out.println("meany= " + meany);
 System.out.println("vary = " + vary);
 System.out.println("ny= " + ny);
 System.out.println("Ho: expected means :");
 System.out.println("mux = " +  mux);
 System.out.println("muy = " +  muy);
 double diffExp = mux-muy;
 double diffObs = meanx - meany;
```

```
 System.out.println("Ho: expected difference of means ="
+diffExp);
 System.out.println("Ho: observed difference of means ="
+diffObs);
 System.out.println("alpha = " + alpha);
}

static private void confIntTwoMeansT(
 double meanx,
 double varx,
 double nx,
 double meany,
 double vary,
 double ny,
 double alpha )
{
 System.out.println( "\nConfidence interval of the " +
" \ndifference between means " +
"when variances are equal");
 double alphaHalf = alpha/2;
 System.out.println( "alpha = " + alpha );
 System.out.println("dfT = nx+ny-2 = " + dfT);
 double tc2 =  p.critical(alphaHalf);
 System.out.println("Critical t  =  tc2 = +/-"
           + tc2);
 double jointDev = Math.pow((varx*(nx-1)
    + vary*(ny-1))/(nx+ny-2), 0.5);
 System.out.println( "jointDev = sj = " + jointDev );
 double r =  tc2 * jointDev;
 System.out.println( "r = tc2 * jointDev" );
 System.out.printf( "r = %10.4f \n", r );
 double limInf2 =  meanx-meany - r;
 double limSup2 =  meanx-meany + r;
 System.out.printf( "limInf = xBar-yBar - r " +
       "= %10.4f \n" , limInf2 );
 System.out.printf( "limSup = xBar-yBar + r " +
       "= %10.4f \n", limSup2 );
}
```

```
private static void tTestTwoMeansEqualVarWork(
 double meanx,
 double varx,
 double nx,
 double meany,
 double vary,
 double ny,
 double mux,
 double muy,
 double alpha )
{
 printData8( meanx, varx,  nx,
 meany,  vary,  ny,
 mux,  muy, alpha,
 "when variances are equal");
 double sj = Math.pow((varx*(nx-1)
    + vary*(ny-1))/(nx+ny-2), 0.5);
 System.out.println("\nsj = Math.pow((var1*(nx-1)"
    + "+var2*(ny-1))/(nx+ny-2), 0.5)" );
 System.out.println("sj = " + sj);
 double den = sj * Math.pow(1/nx + 1/ny, 0.5);
 System.out.println(
 "den = sj * Math.pow(1/nx + 1/ny, 0.5)" );
 System.out.println("den = " + den );
 System.out.printf( "Discrepancy between observed and ");
 System.out.printf( "expected: \n");
 double tExp = ((meanx - meany)-(mux-muy))/den;
 System.out.println(
 "tExp = ((mediax-mediay)-(mux-muy))/den " );
 System.out.println("tExp = " + tExp);
 //Degrees of freedom
 dfT = nx+ny-2;
 System.out.println("dfT = nx+ny-2 = " + dfT);
 //Top abstract class is instructed to take
 //the form of the particular class
 p = t;
 //Truncation of degrees of freedom
 if (dfT > 40) dfT =  40;
```

```
 //Critical values
 double tCriticOneTail = p.critical(alpha);
 double alphaHalf = alpha/2;
 double tCriticTwoTails = p.critical(alphaHalf);
 double pValue = p.pValue(tExp);
 System.out.println("p-value(" +
        tExp + ") = " +
      "p( x > " + tExp + ") \n = " + pValue);
 System.out.printf("t-Critic for TWO TAILS " +
  "= tc2 =  +/-%10.4f \n",  tCriticTwoTails);
 System.out.printf("t-Critic for ONE TAIL  " +
  "= tc1 = +/-%10.4f \n",  tCriticOneTail);
 //Confidence interval
 confIntTwoMeansT( meanx, varx, nx,
           meany, vary, ny, alpha );
 separator();
}


//Comparison of two means. Equal  variances.
public static void tTestTwoMeansEqualVar()
{
 //Example 213, pag. 190.
 double meanx= 62.7;
 double varx =  6.864;
 double nx = 150;
 double meany= 61.8;
 double vary = 6.25;
 double ny= 120;
 double mux= 62.7;
 double muy= 62.7;
 double alpha = 0.02;
 tTestTwoMeansEqualVarWork( meanx, varx,  nx,
 meany,  vary,  ny,
 mux,  muy, alpha );

}

//==== 9: comparing means when variances are different ====
```

```
//t-test to compare the difference between means
//against an expected difference.
//Variances are supposed to be equal.
//Thus, samples are joint together to estimate
//the unique variance.



static private void confIntTwoMeansT2(
 double meanx,
 double varx,
 double nx,
 double meany,
 double vary,
 double ny,
 double alpha )
{
 System.out.println( "\nConfidence interval of the " +
" \ndifference between means " +
"when variances are different");
 double alphaHalf = alpha/2;
 double tc2 =  p.critical(alphaHalf);
 System.out.println( "alpha = " + alpha );
 System.out.println("dfT (truncated) = " + dfT);
 System.out.println("Critical t  =  tc2 = +/-"
            + tc2);
 double  deno = Math.pow(varx/nx + vary/ny, 0.5);
 System.out.println( "jointDev =   " + deno );
 double r =  tc2 * deno;
 System.out.println( "r = tc2 * jointDev" );
 System.out.printf( "r = %10.4f \n", r );
 double limInf2 =  meanx-meany - r;
 double limSup2 =  meanx-meany + r;
 System.out.printf( "limInf = xBar-yBar - r " +
        "= %10.4f \n" , limInf2 );
 System.out.printf( "limSup = xBar-yBar + r " +
        "= %10.4f \n", limSup2 );
}
```

```
private static double tDegreesOfFreedom(
 double varx,
 double nx,
 double vary,
 double ny,
 double alpha )
{
 System.out.println( "\nDegrees of freedom:");
 double num = Math.pow(varx/nx + vary/ny, 2);
 double den1 = (varx/nx)* (varx/nx)/(nx+1);
 double den2 =   (vary/ny)*(vary/ny)/(ny+1);
 double den = den1 + den2;
 dfT =  (int) (num/den -2);
 System.out.println("num  = " +
                "Math.pow(var1/n1 + var2/n2, 2)");
 System.out.println("den1 = (var1/n1)* (var1/n1)/(n1+1)");
 System.out.println("den2 =  (var2/n2)*(var2/n2)/(n2+1)" );
 System.out.println("den = den1 + den2" );
 System.out.println("dfT = num/den -2");

 System.out.println("num  = " + num);
 System.out.println("den1 = " + den1);
 System.out.println("den2 = " + den2);
 System.out.println("den = " + den);
 System.out.println("dfT = " + dfT);
 return dfT;
}


private static void tTestTwoMeansDiffVarWork(
 double meanx,
 double varx,
 double nx,
 double meany,
 double vary,
 double ny,
 double mux,
 double muy,
```

```
 double alpha )
{
 printData8( meanx, varx,  nx,
 meany,  vary,  ny,
 mux,  muy, alpha,
 "when variances are different");
 double  deno = Math.pow(varx/nx + vary/ny, 0.5);
 System.out.println(
"deno =  Math.pow(varx/nx + vary/ny, 0.5)");
 System.out.println( "deno =  " + deno);
 System.out.printf( "Discrepancy between observed and ");
 System.out.printf( "expected: \n");
 double tExp = ((meanx - meany)-(mux-muy))/deno;
 System.out.println(
 "tExp = ((mean-meany)-(mux-muy))/deno " );
 System.out.println( "tExp =   " + tExp );
 dfT = tDegreesOfFreedom( varx, nx, vary, ny, alpha );
 //Truncation of degrees of freedom
 if (dfT > 40) dfT =  40;
 //Top abstract class is instructed to take
 //the form of the particular class
 p = t;

 //Critical values
 double tCriticOneTail = p.critical(alpha);
 double alphaHalf = alpha/2;
 double tCriticTwoTails = p.critical(alphaHalf);
 double pValue = p.pValue(tExp);
 System.out.println("p-value(" +
      tExp + ") = " +
    "p( x > " + tExp + ") \n = " + pValue);
 System.out.printf("t-Critic for TWO TAILS " +
"= tc2 =  +/-%10.4f \n",  tCriticTwoTails);
 System.out.printf("t-Critic for ONE TAIL  " +
"= tc1 = +/-%10.4f \n",  tCriticOneTail);
 //Confidence interval
 confIntTwoMeansT2( meanx, varx, nx,
           meany, vary, ny, alpha );
 separator();
```

```
}


//Comparison of two means. Equal  variances.
public static void tTestTwoMeansDiffVar(
)
{
 //example 215, pag. 192.
 double meanx= 34;
 double varx =  5;
 double nx = 12;
 double meany= 12;
 double vary = 90;
 double ny= 7;
 double mux = 3*meany;
 double muy = meany;
 double alpha = 0.02;
 tTestTwoMeansDiffVarWork( meanx, varx,  nx,
                meany,  vary,  ny,
                mux,  muy, alpha );
}


//==== 8+9a: comparing means + previous F-test====


//Comparison of two means.
//Splits in two cases: equal and different variances.
public static void tTestFullCompTwoMeansWork(
 double meanx,
 double varx,
 int nx,
 double meany,
 double vary,
 int ny,
 double mux,
 double muy,
 double alpha )
```

```
{
 System.out.println("\nFull-fledged Comparison of two" +
                 " means. \nSplits in two cases: " +
                 "equal else different variances.");

 //Test for the equality of variances
 System.out.println("varx = " + varx);
 System.out.println("vary = " + vary);
 System.out.println("\nTest for the equality of variances");
 double R=1;
 double RExp= varx/vary;
 System.out.println("alpha = " + alpha);
 System.out.println("\nExpected relation " +
  "between variances =\nR=" +  R);
 dfFNum = nx-1;
 dfFDen = ny-1;
 if (RExp <1)
  {
RExp = 1/RExp;
double k = dfFNum;
dfFNum = dfFDen;
    dfFDen = k;
  }
 System.out.println("\nRExp = LargeVar/SmallVar");
 System.out.println("RExp = " + RExp);
 double FExp = RExp/R;
 System.out.println("\nFExp = RExp/R");
 System.out.println("FExp = " + FExp);
 //Top abstract class is instructed to take
 //the form of the particular class
 p = F;
 System.out.println("dfFNum = " + dfFNum);
 System.out.println("dfFDen = " + dfFDen);
 //Critical values
 double alphaHalf = alpha/2;
 double FCriticTwoTails = p.critical(alphaHalf);
 System.out.println("alphahalf = " + alphaHalf);
 System.out.println("FCriticTwoTails = " +
 FCriticTwoTails  );
```

```
 if (FExp < FCriticTwoTails )
{
 System.out.println("FExp < FCriticTwoTails " );
 System.out.println("discrepancy is small, "
  + "variances are equal.");
     tTestTwoMeansEqualVarWork( meanx, varx,  nx,
 meany,  vary,  ny,
 mux,  muy, alpha );
    }
  else
   {
 System.out.println("FExp > FCriticTwoTails " );
 System.out.println("discrepancy is great, "
  + "variances are different." );
     tTestTwoMeansDiffVarWork( meanx, varx,  nx,
               meany,  vary,  ny,
               mux,  muy, alpha );
   }
}


//Comparison of two means.
//Splits in two cases: equal and different variances.
public static void tTestFullCompTwoMeans()
{
 //Ej 214 pag 191:equal variances
 double meanx= 4;
 double varx =  3.2;
 int nx = 15;
 double meany= 3.5;
 double vary = 4;
 int ny= 20;
 double mux= 0;
 double muy= 0;
 double alpha = 0.05;



 //Ej 216 pag 193: different variances
```

```
 /*double meanx= 3;
 double varx =  1;
 int nx = 7;
 double meany= 1;
 double vary = 0.0001;
 int ny= 10;
 double mux= 0;
 double muy= 0;
 double alpha = 0.05;*/
 tTestFullCompTwoMeansWork( meanx, varx,  nx,
         meany,  vary,  ny,
         mux,  muy, alpha );
 }

//==== 8+9b: comparing means for 2 lists of data ====


//Comparison of two means.
//Splits in two cases: equal and different variances.
//Lists of data
public static void tTestFullCompTwoMeansLists()
{
 System.out.println("Test to compare means " +
"of two lists of data");
 double    DataVect1[]= {1,2,3};
 double    DataVect2[]= {4,5,6};
 double alpha = 0.05;
 meanVarVector(DataVect1);
 meanVarVector(DataVect2);
 //A table is declared
 Table T1 =  a.new   Table(DataVect1);
 int nx = T1.length;
 double meanx = T1.mean(T1);
 double varx = T1.var(T1);
 //A table is declared
 //A table is declared
 Table T2 =  a.new   Table(DataVect2);

 int ny = T2.length;
```

```
 double meany = T2.mean(T2);
 double vary = T2.var(T2);
 double mux=1;
 double muy = 1;
 tTestFullCompTwoMeansWork( meanx, varx,  nx,
       meany,  vary,  ny,
       mux,  muy, alpha );
}

//==== 8+9c: comparing means for 2 tables =====
//==== of absolute frequencies ==============

public static void tTestFullCompTwoMeansFTWork(
double FreqTX[][],   double FreqTY[][],
double mux, double muy,    double alpha)
{
 //A table is declared
 FreqTable TX =  a.new   FreqTable(FreqTX);
 System.out.println("\nTable of absolute frequencies X.");
 meanVarFreqTable(FreqTX);
 double meanx = TX.mean(TX);
 double varx = TX.var(TX);
 int nx =  (int)  TX.sum(TX,1);
 FreqTable TY =  a.new   FreqTable(FreqTY);
 System.out.println("\nTable  of absolute frequencies Y.");
 meanVarFreqTable(FreqTY);
 double meany = TY.mean(TY);
 double vary = TY.var(TY);
 int ny =  (int) TY.sum(TY,1);
 tTestFullCompTwoMeansWork( meanx, varx,  nx,
     meany,  vary,  ny,
     mux,  muy, alpha );
}

//Comparison of two means.
//Tables of absolute frequencies
public static void tTestFullCompTwoMeansFT()
{
 System.out.println("Comparison of two means." +
```

```
   "\nSplits in two cases: equal and different variances."+
   "\nInput: two tables of  frequency tables.");

 //Problem 2, partial 3, version 5, pag. 165.
/* double    FreqTX[][] =
        {{3, 3},
         {4, 3},
         {5, 2},
         {6, 2},
         {7, 1},
         {8, 1}};*/

 double    FreqTX[][] =
                       {{3, 3},
                        {4, 3},
                        {5, 2},
                        {6, 3},
                        {7, 2},
                        {8, 2},
                        {9,3}};

 double    FreqTY[][] =
        {
         {5, 1},
         {6, 3},
         {7, 1}};
 double alpha = 0.05;
 double muy = 5;
 double mux = 5;
 tTestFullCompTwoMeansFTWork( FreqTX, FreqTY,
                     mux, muy, alpha);
}

//==== 8+9d: comparing means for 2 tables =====
//==== of cumulative frequencies ==============


//Comparison of two means.
//Splits in two cases: equal and different variances.
```

```
//Tables of cumulative frequencies


//Comparison of two means: CF
//Splits in two cases: equal and different variances.
//Input: two tables of cumulative frequencies.
public static void fullCompTwoMeansCF()
{
System.out.println("Comparison of two means." +
  "\nSplits in two cases: equal and different variances."+
 "\nInput: two tables of cumulative frequencies.");

 double    cumFreqTX[][] =
            {{14,0},
         {15,15},
         {16,20},
         {17,25}};

 double    cumFreqTY[][] =
             {{3, 0},
              {4, 3},
              {5, 7},
              {6, 12},
              {7, 17},
              {8, 19},
              {9,22}};

 double alpha = 0.05;
 double muy = 20;
 double mux = 12;
 System.out.println("\nTable X. " +
 "\nClass marker and its cumulative frequency");
 printMatrix( cumFreqTX, "Cumulative Table X");
 int m = cumFreqTX.length;
 double FreqTX[][] = new double[m][2];
 FreqTX[0] = cumFreqTX[0];

 for(int i = 1; i < m;  i++)
  {
```

```
   FreqTX[i][0] = cumFreqTX[i][0];
   FreqTX[i][1] = cumFreqTX[i][1]-cumFreqTX[i-1][1];
  }


 System.out.println("\nTable Y. " +
 "\nClass marker and its cumulative frequency");
 printMatrix( cumFreqTY, "Cumulative Table Y");
 int n = cumFreqTY.length;
 double FreqTY[][] = new double[n][2];
 FreqTY[0] = cumFreqTY[0];
 for(int i = 1; i < n;  i++)
  {
   FreqTY[i][0] = cumFreqTY[i][0];
   FreqTY[i][1] = cumFreqTY[i][1]-cumFreqTY[i-1][1];
  }
 tTestFullCompTwoMeansFTWork( FreqTX, FreqTY,
         mux, muy,   alpha);
}

//==== 10: comparing means for paired data ====

//t-test to compare the difference between means
// of paired data against an expected difference.
//Data

//This inner class defines a new type: TablePairedData.
//It converts array Pair (two dimensional) into
//a class or object, ready for reuse.
private  class TablePairedData
{
//The class TablePairedData has two main elements:
//1) The titles of  the columns, whose length
//gives the number of columns.
//2) A matrix, whose length is the number of rows.

String[] Titles = { "x",  "y",   "D= x-y", "Dˆ2"};
double M[][] = new double[2000][Titles.length];
int nRows;
```

```
//The input of Table is defined here
//by means of an initialization procedure
//that is called constructor.
//The natural input is the matrix of paired data
//which is consigned in a matrix Pairs with 2 columns.
TablePairedData(double[][] Pairs )
 {
   System.out.println("t-test to compare " +
                      "means for paired data");
nRows = Pairs.length;
   //The matrix M of Table is initialized
   //Raw data go in   columns 0 and 1.
   //Squares in the second.
   for(int i = 0; i < nRows; i++)
   {
 M[i][0] = Pairs[i][0];
 M[i][1] = Pairs[i][1];
 M[i][2] = M[i][0] - M[i][1];
 M[i][3] = M[i][2] * M[i][2];
   }
  }

//Data of column k are summed.
//The input is a table, the output is of type double.
private  double sum(TablePairedData  T, int k)
 {
  double sum = 0;
  for(int i = 0; i < T.nRows; i++)
  sum  = sum +  T.M[i][k];
  return sum;
 }

private  void printTable(TablePairedData T)
 {
  System.out.println("\nExtended matrix to " +
        "calculate the variance of paired data:");
  System.out.printf("%n %10s %10s %10s %10s",
            T.Titles [0], T.Titles [1],
             T.Titles [2],T.Titles [3]);
```

```
  for ( int i = 0; i < T.nRows; i++ )
 System.out.printf(
"%n %10.3f %10.3f %10.3f %10.3f ",
T.M[i][0], T.M[i][1], T.M[i][2], T.M[i][3] );
  System.out.println( );
 }
}//end of inner class


//Confidence interval for paired data
static private void confIntMeansPairedT(
double DBar, int  n, double sD,   double alpha )
{
System.out.println( "\nConfidence interval of the " +
" \ndifference between means " +
"of paired data");
double alphaHalf = alpha/2;
System.out.println("dfT (truncated) = " + dfT);
double tc2 =  p.critical(alphaHalf);
System.out.println( "alpha = " + alpha );
System.out.println("Critical t  =  tc2 = +/-"
           + tc2);
System.out.println( "DBar =   " + DBar );
System.out.println( "sD = " + sD );
double r =  tc2 * sD/ Math.sqrt(n);
System.out.println( "r = tc2 * sD/ Math.sqrt(n)" );
System.out.printf( "r = %10.4f \n", r );
double limInf2 =  DBar - r;
double limSup2 =  DBar + r;
System.out.printf( "limInf = xBar-yBar - r " +
        "= %10.4f \n" , limInf2 );
System.out.printf( "limSup = xBar-yBar + r " +
        "= %10.4f \n", limSup2 );
}


//t-test for the comparison of means of paired data.
private static void tTestPairedDataWork(
       double[][] Pair,   double alpha, double DeltaBar)
```

```
{
 //A table is declared
 TablePairedData T =  a.new   TablePairedData(Pair);
 T.printTable(T);
 int n = Pair.length;
 double sumx = T.sum(T, 0);
 double sumy = T.sum(T, 1);
 double sumD = T.sum(T, 2);
 double sumD2 = T.sum(T, 3);
 System.out.println("Sums");
 System.out.printf(
"%10.3f %10.3f %10.3f %10.3f ",
sumx, sumy, sumD, sumD2 );
 double DBar = sumD / n;
 System.out.println( "\n\nn = Par.length");
 System.out.println( "n = "+n);
 System.out.println( "sum(x-y) = sumD = " + sumD);
 System.out.println(
 "DBar = Observed mean difference = sumD / n");
 System.out.println( "DBar =   "+ DBar);
 // Let us calculate the variance
 System.out.println( "\nVariance:");
 System.out.println( "sumDˆ2= "+ sumD2);
 System.out.println(
      "var =  (n* sumD2 - sumD*sumD) / (n * (n-1))" );
 double var =  (n* sumD2 - sumD*sumD) / (n * (n-1));
 System.out.println( "Variance = "+ var);
 double sD = Math.pow( var, 0.5);
 System.out.println( "sD = Deviation = Math.pow(var, 0.5)");
 System.out.println( "sD =  "+ sD);

 //Expected difference between means
 System.out.println(
 "DeltaBar = Expected mean difference = "+ DeltaBar);
 double  tExp = (DBar - DeltaBar)/(sD/Math.pow( n, 0.5));
 System.out.println(
     "tExp = (DBar - delta)/(des/Math.pow( n, 0.5)) ");
 System.out.println(
   "Discrepancy between observed and expected = tExp");
```

```
 System.out.println("tExp = " + tExp);
 dfT = n-1;
 System.out.println("Degrees of freedom = " + dfT);
//Truncation of degrees of freedom
 if (dfT > 40) dfT =  40;
 //Top abstract class is instructed to take
 //the form of the particular class
 p = t;

 //Critical values
 double tCriticOneTail = p.critical(alpha);
 double alphaHalf = alpha/2;
 double tCriticTwoTails = p.critical(alphaHalf);
 double pValue = p.pValue(tExp);
 System.out.println("p-value(" +
      tExp + ") = " +
    "p( x > " + tExp + ") \n = " + pValue);
 System.out.printf("t-Critic for TWO TAILS " +
"= tc2 =  +/-%10.4f \n",  tCriticTwoTails);
 System.out.printf("t-Critic for ONE TAIL  " +
"= tc1 = +/-%10.4f \n",  tCriticOneTail);
 //Confidence interval
 confIntMeansPairedT( DBar, n, sD, alpha );
 separator();
}


//Data input to a t-test for means of  paired data.
private static void tTestPairedData()
{

//Example 217, pag. 195.
double    Pair[][] =
           { {20,19},
         {23,17},
         {30,20},
         {27,23},
         {25,15},
         {21,19}};
```

```
double alpha = 0.05;
//Expected mean difference
double DeltaBar = 18.8;
tTestPairedDataWork(Pair, alpha, DeltaBar);

}



//==== 11: chi2 -test for factor independence ====

public static void printContTable(double M[][],
                 int nCol,  int nRows, String s)
{
 //Totals
 double n = 0;
 for(int i = 0; i< nRows; i++)
 for(int j = 0; j< nCol; j++)
   n = n + M[i][j];

 double totalCol[] = new double[10];
 double totalRow[] = new double[10];

 //Sums by rows and columns
 for(int i = 0; i< nRows; i++)
  {
totalRow[i] = 0;
for(int j = 0; j< nCol; j++)
  totalRow[i]   = totalRow[i] + M[i][j];
  }
 for(int j = 0; j< nCol; j++)
  {
totalCol[j] = 0;
for(int i = 0; i< nRows; i++)
totalCol[j]   = totalCol[j] + M[i][j];
  }

 System.out.println("" + s + " and totals");
 for(int i = 0; i< nRows; i++)
```

```
  {
for(int j = 0; j< nCol; j++)
System.out.printf("%10.4f   ",  M[i][j]  );
System.out.printf("| %10.4f   ", totalRow[i]  );
System.out.println();
  }

 for(int j = 0; j<= nCol; j++)
  System.out.printf( "%10s   ", "---");
 System.out.printf( "%n");

 for(int j = 0; j< nCol; j++)
  System.out.printf("%10.4f   ",  totalCol[j]   );
 System.out.printf("| %10.4f   ", n );
 //System.out.println( "\nGreat Total = n = " + n);
}

//A chi2-test for factor independence is run
//over a contingency table.
public static void factorIndependenceWork(double Obs[][],
    int nCol,  int nRows, double alpha, String s)
{
 System.out.println(s);

 //Totals
 double n = 0;
 for(int i = 0; i< nRows; i++)
  for(int j = 0; j< nCol; j++)
   n = n + Obs[i][j];

 double totalCol[] = new double[10];
 double totalRow[] = new double[10];

 //Sums by rows and columns
 for(int i = 0; i< nRows; i++)
  {
 totalRow[i] = 0;
  for(int j = 0; j< nCol; j++)
  totalRow[i]   = totalRow[i] + Obs[i][j];
```

```
  }
 for(int j = 0; j< nCol; j++)
  {
 totalCol[j] = 0;
for(int i = 0; i< nRows; i++)
totalCol[j]   = totalCol[j] + Obs[i][j];
 }

 printContTable(Obs, nCol,  nRows, "Observed");

 //Expected under factor independence

 double  Exp[][]  = new double[10][10];
 for(int i = 0; i< nRows; i++)
  for(int j = 0; j< nCol; j++)
  Exp[i][j]  = totalRow[i] * totalCol[j]/n;
 //Errors, cell by cell
 System.out.println("\n\nExp[i][j]  " +
              "= totalRow[i] * totalCol[j]/n");
 printContTable(Exp, nCol,  nRows,
 "Expected under factor independence");
 System.out.println( "\n\nCalculation of discrepancy " +
               "between observed and expected:" );
 System.out.println("Errors, cell by cell" +
              " = Obs[i][j]*Obs[i][j]/Exp[i][j]");
 System.out.println("Observed, " +
                "expected and errors cell by cell");
 double bigSum = 0;
 for(int i = 0; i< nRows; i++)
  for(int j = 0; j< nCol; j++)
   {
double errorCell = Obs[i][j]*Obs[i][j]/Exp[i][j];
bigSum  = bigSum + errorCell;
    System.out.printf("%10.4f     %10.4f     %10.4f%n",
      Obs[i][j], Exp[i][j],errorCell);
    }
 System.out.println("Sum of errors cell by cell " +
              "= bigSum = "  + bigSum);
 System.out.println("n = " + n);
```

```
 double chi2Exp = bigSum-n;
 System.out.printf( "Discrepancy between" +
  " observed and expected \n= chi2Exp = " +
  "bigSum-n = %10.4f%n%n", chi2Exp);
 dfChi2 = (nCol-1)*(nRows-1);
 //Top abstract class is instructed to take
 //the form of the particular class
 p = chi2;
 p.critManager(alpha);
 separator();
}

public static void factorIndependence()
{
 //Example 231, pag. 210
 double  Obs[][]  = {
{4,3,2},
            {3,2,1},
            {2,1,1}};
 int nCol = 3;
 int nRows = 3;
 double alpha = 0.05;
 String s = "A chi2-test for factor independence";
 factorIndependenceWork(Obs,  nCol, nRows, alpha, s);
}

//==== 12: chi2 -test for proportion homogeneity ====

public static void chi2TestPropHomogeneity()
{
 //Example 232, pag. 212
 double  Obs[][]  = {
{7,6,1},
            {15,18,20}};
 int nCol = 3;
 int nRows = 2;
 double alpha = 0.05;
 String s = "A chi2-test for proportion homogeinity";
 factorIndependenceWork(Obs,  nCol, nRows, alpha, s);
```

```
}

}//End of main class
```

Bibliography

[1] Ammermann D (1985) Chromatin diminution and chromosome elimination : mechanism and adaptive significance. In The Evolution of Genome Size, edited by Cavalier -Smith, J. Wiley & Sons Ltd., pp 427–442.

[2] Arneodo A, Aubenton Y, Bacry E, Graves P, Muzy J, Thermes C, (1996), Wavelet based fractal analysis of DNA sequences , *Physica D* 96 291–320.

[3] Cavalier-Smith T,(1985) Introduction : the Evolutionary Significance of Genome Size. In *The Evolution of Genome Size*, edited by Cavalier -Smith, J. Wiley & Sons Ltd., 1–36.

[4] Developer (2011)
http://www.developer.com/java.
Verified 23 Jun 2011.

[5] Eclipse (classic) (2011)
http://www.eclipse.org/downloads/.
Verified 23 Jun 2011.

[6] Dexter M(2011)
http://sourceforge.net/projects/eclipsetutorial/
Verified 23 Jun 2011.

[7] Foote B, Yoder J (1995) Evolution, architecture and metamorphosis
http://www.laputan.org/metamorphosis/metamorphosis.html
Verified 24 Jun 2011

[8] Gregory, T.R. (2005). Animal Genome Size Database.
http://www.genomesize.com.
Verified 24 Jun 2011.

[9] Holzner S (2004) Eclipse, O'Reilly Media, Sebastopol, Ca.

[10] Ihmels J, Collins S, Schuldiner M, Krogan N,Weissman J (2007) Backup without redundancy: genetic interactions reveal the cost of duplicate gene loss. *Molecular Systems Biology* 3: 86 doi:10.1038/msb4100127.

[11] Lee J (2003) Molecular biology: Complexity of gene and pseudogene. *Nature* 423, 26–28 — doi:10.1038/423026a

[12] Oracle (2011a) Free Java download
http://java.com/en/download/index.jsp.
Verified 23 Jun 2011.

[13] Oracle (2011b) Java Developer Tutorials and Training.
http://www.oracle.com/technetwork/java/index-jsp-135888.html.
Verified 23 Jun 2011.

[14] Swartz Fred (2007) Java Basics, Notes-Java.
http://www.leepoint.net/notes-java/index.html.
Verified 23 Jun 2011.

[15] Zamenhof S, Eichhorn H, (1962), Study of Microbial Evolution through Loss of Biosynthetic Functions. Establishment of "Defective" Mutants. *Nature.* 216: 456–458.

Index

computer, 1
hardware, 1

alphabet, 1

bug, 7

C-paradox, 94
class, 2
code, 1
code of interpretation, 1
compilation, 7
console, 6
copy, 5

data
    quantitative, 11
data list, 11
debugging, 2, 8
delete, 5
developer, 1
distribution
    z, 59

Eclipse, 2, 3
encapsulation, 164
error
    correction , 7
event
    extreme, 60
evolJava community, 5
evolution, 2
evolutionary inertia, 164
evolvable software, 163

functional style, 163

gossipless style, 163

hypothesis
    alternate , 60
    null , 60

Java, 2
    installing, 2

language, 1

mean or average, 11
method, 2

OOP style, 163

p-value, 59
pasting, 5
printf, 14, 167
problem of granulation, 164
program, 1
programming language, 1
project, 4
projects, 2

robot, 1

science, 99
scientific method
    naked , 60
select all, 5

self-test, 14
semantic content, 1
significance level, 59
Software, 1
software is known by its fruits, 8
standard deviation, 11
Statistical inference, 99
style
    entangled OOP, 93
    functional, 18
    gossipless, 24
    motherland, 15
System.out.println, 6

table, 167
    cumulative frequency , 28
    frequency, 27
tails
    one, 59
    two, 59
test, 60
toolbox, 33
transparent style, 24, 163

value
    lower critical , 60
    upper critical , 60
variance
    sample, 11
verbal instruction, 1

window
    editor , 9
workspace, 3