**Department of Economics**

# z-Tree 4.1

## Tutorial and Reference Manual

Urs Fischbacher, Thurgau Institute of Economics at the University of Konstanz
Katharine Bendrick, Southern Connecticut State University
Stefan Schmid, University of Zurich

# Table of Contents

# List of Figures

# List of Tables

# Tutorial

# 1 About z-Tree

The z-Tree program was initially developed at the University of Zurich by Urs Fischbacher. Now, it is a joint project of the University of Zurich, the Thurgau Institute of Economics and the University of Konstanz with Urs Fischbacher und Stefan Schmid working on the program. It is specially designed to enable the conducting of economic experiments without much prior experience. It consists, on the one hand, of z-Tree, the "Zurich Toolbox for Readymade Experiments", and, on the other hand, of z-Leaf, the program used by the subjects.

In z-Tree, you can define and conduct experiments. You can program a broad range of experiments with z-Tree, including public goods games, structured bargaining experiments, posted-offer markets or double auctions. The programming of z-Tree requires a certain amount of experience. Thereafter, the effort required for conducting experiments is minimal: An experimenter with some experience in z-Tree can program a public goods game in less than an hour and a double auction in less than a day.

On performance: In Zurich, z-Tree has been used for almost all experiments that are conducted with computers. We started with a lab containing 26 PCs with 486er processors and 16 MB RAM which are connected on an Ethernet. The program always worked efficiently in this configuration. For more complicated programs, a good performing computer helps - in particular for z-Tree.

The manual of z-Tree consists of two parts, the tutorial and the reference manual. The tutorial can be read sequentially. It starts in Chapter 2, *Introduction* with a guided tour in which you learn the basic elements of z-Tree programming. This chapter is concluded with a detailed explanation of how to set up the environment to test z-Tree on a single computer. In Chapter 3, *Definition of Treatments* you learn how to program experiments. Chapter 4, *Questionnaires* can be omitted for the first reading. Chapter 5, *Conducting a Session* is essential as soon as you conduct the first experiment. It explains the normal procedure of an experimental session as well as how to deal with emergencies such as computer crashes. Chapter 6, *Installation and lab architecture* explains how z-Tree can be installed in a lab.

z-Tree is very flexible. Nevertheless, it may occur that you wish to realize something that is not covered by the program. On the z-Tree website at https://www.ztree.uzh.ch/, you will find tips and tricks. If you still feel something is missing or if you should find an error in the program or in the manual, please send an email to ztree@econ.uzh.ch [mailto:ztree@econ.uzh.ch].

We would like to thank the users of z-Tree for their patience with the program and for their suggestions on how to improve the program, in particular Vital Anderhub, Armin Falk, Ernst Fehr, Simon Gächter, Florian Knust, Oliver Kirchkamp, Andreas Laschke, Stefan Palan, Martin Strobel, and Jean Robert Tyran. We would also like to thank Silvana Christ, Alan Durell, Armin Falk, Christina Fong, Cornelia Schnyder, Omar Solanki, and Beatrice Zanella for helping to present the program in this manual. Omar Solanki translated the first German manual into English.

### 1.1 Version History

– Version 4.1.0-1, 17.8.2018

   – Fix of typos in 8.3.38 and 8.3.40

– Version 4.1.0, 9.8.2018

   – Resized figures with item dialog.

   – Added info that the layout can be left out for string output.

   – Added better information about GameSafe export possibility.

– Version 4.0.31, 21.6.2018

   – Changed figures with item dialog.

   – Added documentation of functions stringinsert, stringdelete, replaceall, and optionswitch.

### 1.2 Styles used in this manual

Programs and code snippets are presented in a `monospaced` font. Text that should be replaced with user-supplied values or by values determined by the context is shown *slanted*.

```
if( condition ) {
    statements
}
else {
    statements
}
```

In the example above, "if", "else", and the parentheses have to be written exactly as shown. "condition" and "statements" have to be replaced by the appropriate text.

Labels in dialogs are presented in the text in a sans serif font on a gray background: Number of subjects

Entries into fields are shown in a box: `0.1`

Menus and menu commands look like File → Save As…

Keys on the keyboard appear like **Enter** or **Ctrl+c**

Stage tree elements are written in a sans serif font: Active screen

### 1.3 How to use this manual

The manual is divided into a tutorial and a reference manual. The tutorial starts with a guided tour with a step by step explanation of two basic treatments, e.g. experiment parts. Up to Section Section 3.7, "Posted offer markets", the sections depend on each other. Later the sections can be read independently, according to your needs.

# 2 Introduction

## 2.1 Terms

In a non-computerized experiment there is one or more *experimenter* and a number of *subjects*. The latter communicate with one another through the experimenter. In a computerized experiment this communication takes place through the computer. The computer operated by the experimenter is called the *experimenter PC*. The computers operated by the subjects are called *subject PCs*. The program the experimenter works with is called "z-Tree"; it is the *server program* or in short, the server. The program the subjects work with is called "z-Leaf"; it is the *client program* or in short, the client.
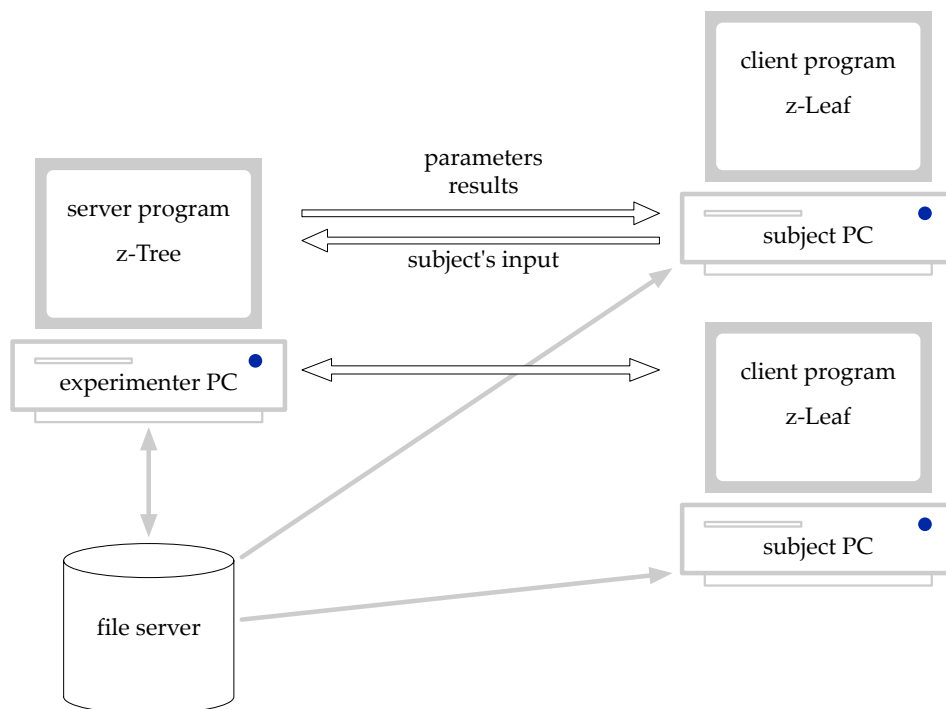
Figure 2.1. Client-server architecture of z-Tree

Be careful not to confuse the server program with the file server. The latter is used to save the programs, the data and the results of an experiment. A file server is not mandatory for conducting an experiment since TCP/IP is used for communication between z-Tree and z-Leaf. As soon as the clients have established contact with the server, communication between server and clients takes place directly, not via the file server. However, a file server is useful for storing the experimental data and it does facilitate start-up. When starting, z-Tree can write its own IP address into a file on the file server. This allows the clients to find out which computer the server was started on. This makes it easily possible to use different computers as experimenter PCs without having to inform the clients explicitly.

By *session* we mean the events that occur in the time span between the arrival of the subjects and the moment they have received payment. A set of corresponding sessions constitutes an *experiment*. A *treatment* is a part of a session. The definition of each treatment is stored in one file. A treatment is a type of z-Tree document. How treatments are defined is explained in Chapter 3, *Definition of Treatments*. Each session consists of one or more treatments and questionnaires. Questionnaires are also a type of z-Tree document and can be freely defined. Questionnaires will be explained in Chapter 4, *Questionnaires*.

In Section 2.2, "A very simple game", we present a simple non-interactive experiments and in Section 2.4, "Step by step to the simple game" we present in a guided tour how to program it. In Section 2.5, "Testing the treatment", we show how to test it. The purpose of these sections is to get a first impression of how to program with z-Tree.

## 2.2 A very simple game

We present how to design a simple non-interactive experiment. This experiment shows how to give information to the subjects and how to request input.

In this experiment, subjects have to enter a number between 0 and 5. This number is multiplied with a factor of 2, and this amount will be paid out to the subjects. Subjects will be shown two screens.
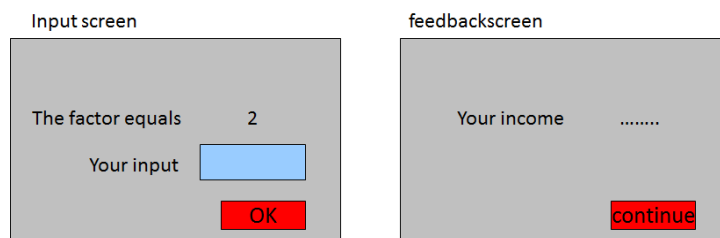


Figure 2.2. The screen layout of the simple game

On the first screen, the input screen, they will be informed about the factor, and they can make the input. On the second screen, the feedback screen, they will be informed about their payoff. It is a good practice to first sketch the screen layout.

## 2.3 Basic concepts

Before we continue, we will define some of the key concepts of z-Tree. Experiments are defined in `treatments`, which are stored as files. The treatments consist of a `stage tree`, which contains all the information about a treatment. It is called stage tree because it consists of a sequence of stages and because the stages contain other items, which can contain other items and so forth. This structure is displayed in a tree structure. Figure Figure 2.3, "The stage tree of the simple game" shows the stage tree of the simple game.
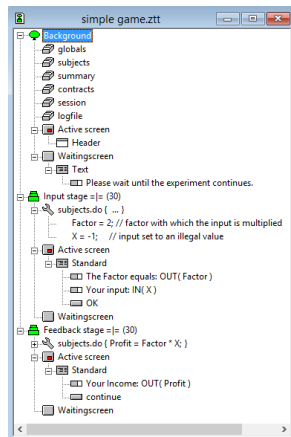
Figure 2.3. The stage tree of the simple game

It shows the two stages, the `Input stage` and the `Feedback stage`. Each stage contains two screen, the `active screen` and the `waiting screen`. We put all the relevant information into the active screen. As the name says, the waiting screen is only used when the subjects wait for other subjects. Into the screen, we put boxes. In our case, we just need one standard box. This box displays information line by line. What information is shown or requested is defined in items.

All the information that is shown and read in as input is stored in variables. These variables have a name. In our example we name the factor `Factor`, and the input X. The resulting income is called `Profit`. This is a general variable that z-Tree uses to determine the payment. The items provide the interface between these variables and the display.

## 2.4 Step by step to the simple game

We are now ready to start and define the treatment step by step. This means that we fill the stage tree with the appropriate stage tree items.
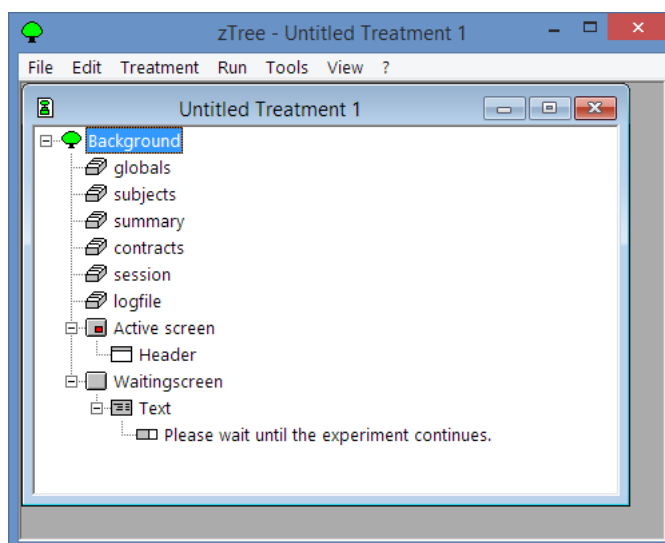


Figure 2.4. z-Tree starts with an untitled treatment

First, we start ztree.exe. The program opens with an empty treatment, in which the background item is selected. First, we add two stages for the two steps.

– If the Background is not yet selected, select it by la left click on the tree item.

– Choose from the menu Treatment → New Stage….

– In this dialog, one can choose some options. In our treatment, we do not have to change the default options. We only change the name to `Input Stage` and close the dialog with the OK button.

– The stage is automatically produced with the two screens.

– The input stage should now be selected. If not, select it.

– Choose Treatment → New Stage….

– In the dialog that pops up, change the name to `Feedback Stage` and close the dialog with the OK button.

Now we define the variables. We select the input stage item, and choose Treatment → New Program…. The dialog for the program pops up.
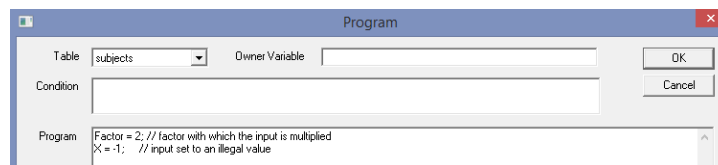


Figure 2.5. Program dialog

We write the definition of the variables into the big Program field.

```
Factor = 2; // factor with which the input is multiplied
X = -1;     // input set to an illegal value
```

The first line reads as: the variable `Factor` gets the value of 2. The semicolon finishes the command and the two slashes define a comment. It is ignored by the program, but can contain useful information for the programmer. In the second line, we set X to -1, which is a value that cannot be the result of an entry by a subject. So if we observe a value of -1 in the data, we know that no entry has been made. Note that z-Tree is case sensitive. This means that ab, Ab, aB, and AB are all different variables.

Now we define what will be shown to the subjects in the input stage. Select the active screen in the input stage. Choose Treatment → Standard Box; it is contained in the submenu Treatment → New Box. There is nothing to change in the dialog that pops up. We put two "items" into this standard box. The first item displays the factor. Choose the menu Treatment → New Item. In the item dialog the following entries have to be made.
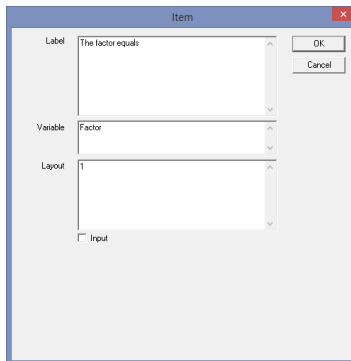
Figure 2.6. Dialog for item factor

The text `The Factor equals` entered in the Label is shown left of the value. The Variable contains the name `Factor` of the variable. The value of the variable, i.e., "2", will be shown of the right hand side. The entry of `1` in the Layout field means that the variable is a multiple of one and displayed without decimals. The entries are confirmed using the OK button.

The entries for the second item, the input item, look like in the following figure.
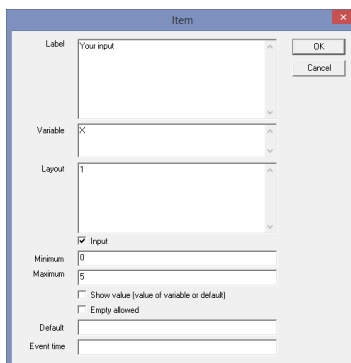


Figure 2.7. Dialog for item X

The Label `Your input` is also shown as text on the left hand side. The Variable contains `X`, which means that the subject's entry is stored in the variable X. The entry of `1` in the Layout means that the variable is a multiple of one and only integer values are accepted. The item defines an input item because the Input check box is checked. In this case, additional entries can be made. Minimum and Maximum are mandatory and specify the lower and upper limits of the entry. In our example, the entered number must lie between 0 and 5, including these values. This implements automatically a range check. The other fields are not relevant for now. Also this dialog has to be concluded with the OK button.

Finally, we have to define a button for the subjects. Choose Treatment → New Button. In the dialog, we accept the default `OK` as a name for this button.

The two items and the button define how the screen of the subjects will look like.

Now, we define the Feedback stage. First, we have to create a program. Choose Treatment → New Program and enter the following program.

```
Profit = Factor * X;
```

In the active screen in the Feedback stage, we add again a Standard box. In this box we add an item with Label `Your income`, Variable `Profit`, and Layout `1`. Finally, we add a button. We give it the Name `continue`. This makes clear that the subject does not have to confirm an entry.

## 2.5 Testing the treatment

Before testing the treatment, it makes sense to save it using the command File → Save as.

In order to test the treatment, you start zleaf.exe. Using **Alt+tab**, you move back to z-Tree. Now the Run Menu gets into focus. Choose Run → Clients' table. You should see one client connected. It is named as your computer. Now, choose Run → Start Treatment. With **Alt+tab**, move to zleaf.

Now you (the subject) can make an entry. If you enter values outside of the defined range, you will get a error message. By default, this message is in German. We will show in Section Section 2.7, "Using another language than German" how to change it. When you have entered a value and clicked on the "OK" button, you will get to the feedback screen. You do not automatically proceed. When the time expires, you only get a warning. Thus, you are forced to make an entry.

The feedback screen will end when you have pressed "continue" or when the time is over. Since no entry has to be made automatic continuation is not a problem. When you have left the feedback screen, you see the waiting screen. This waiting screen will be visible until you start this or another treatment.

You should see screens like the ones in the following figure.
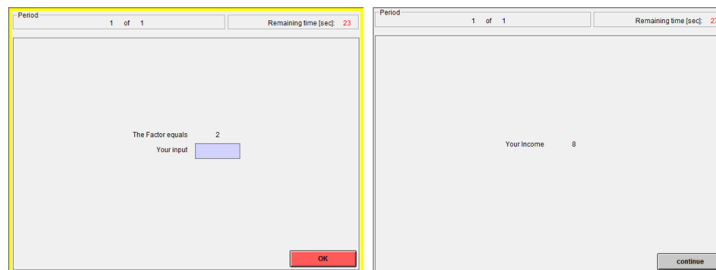


Figure 2.8. Screens of the simple game

At the end of the treatment, you see the message of the waiting screen. It will stay there until you start a new treatment. This is the normal procedure in an experiment. One treatment after the other treatment is started. The treatments are stored in files and can be saved and opened.

## 2.6 Testing the treatment with more subjects

In this section, we will show how to test the treatment with more that one subject. In order to do this we save the treatment. Then, we close z-Tree and z-Leaf because we cannot conduct treatments with different number of subjects. Then, we restart z-Tree. We open again the treatment.

Testing allows you to find the mistakes in a treatment. Most errors can also be found with a reduced number of subjects. It is, therefore, a good strategy to try a treatment first with a reduced number of subjects because it is much easier. In particular, you only have to make the input for a reduced number of subjects. In general, it is also sufficient to try out only one period. We will now test our treatment

with two subjects and one period. First, you have to change the parameters in the Background dialog accordingly. In this dialog, you set in particular the number of subjects.

To test a treatment, you have two possibilities. If you work in a lab you can start z-Tree and one z-Leaf per subject each on a separate computer. The easiest way to do this is first to start z-Tree from a directory on a file server and then to start z-Leaf from the same network directory. On each computer where you start z-Leaf, the starting screen of z-Leaf appears.

If you want to test the treatment outside of the lab, you can also run z-Tree and several z-Leaves on a single computer. You just have to give the different z-Leaves different names. You can achieve this by creating shortcuts to zleaf.exe (in explorer: File → Create Shortcut). For each shortcut, you open the properties dialog, click on the shortcut tab and append the text /name *Yourleafname* to the target field (see Figure 2.9, "Windows shortcut dialog"). There must be a space before and after /name and there may be no space after the slash (/). *Yourleafname* can be any name. It must be different for the different shortcuts. You can then start the z-Leaves by double clicking on the shortcuts. You can switch between different programs with the **Alt+Tab** key combination.
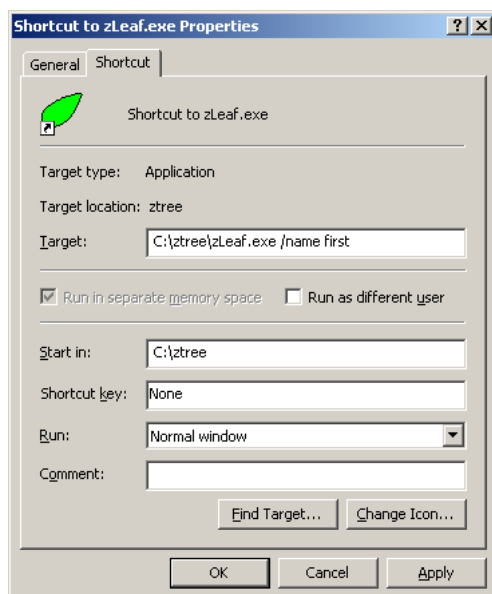


Figure 2.9. Windows shortcut dialog

You can also start several z-Leaves with a batch file. Open a text editor and write the text
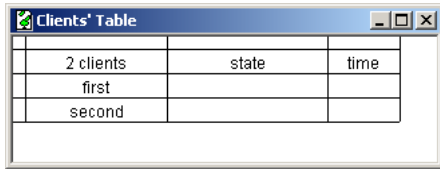
```
start zleaf.exe /name first
start zleaf.exe /name second
```

Save this file with a name ending with ".bat" or ".cmd", e.g. call the file "start2leaves.bat".[1] Put this file into the directory of zleaf.exe and run the batch file (by double clicking). This second method is particularly useful if you have to start many z-Leaves at once.

No matter whether you start z-Leaf on the computer where z-Tree runs or whether you start it on another computer in the lab, you can check how many z-Leaves are actually connected with the z-Tree you are currently running. You do the following: In z-Tree, you choose Run → Clients' Table. When the

---

[1] In order to change the extension of a file in the Windows explorer, you have to uncheck the option "Hide extensions of known file types". You find this option in the "folder option" dialog in the tab "view".
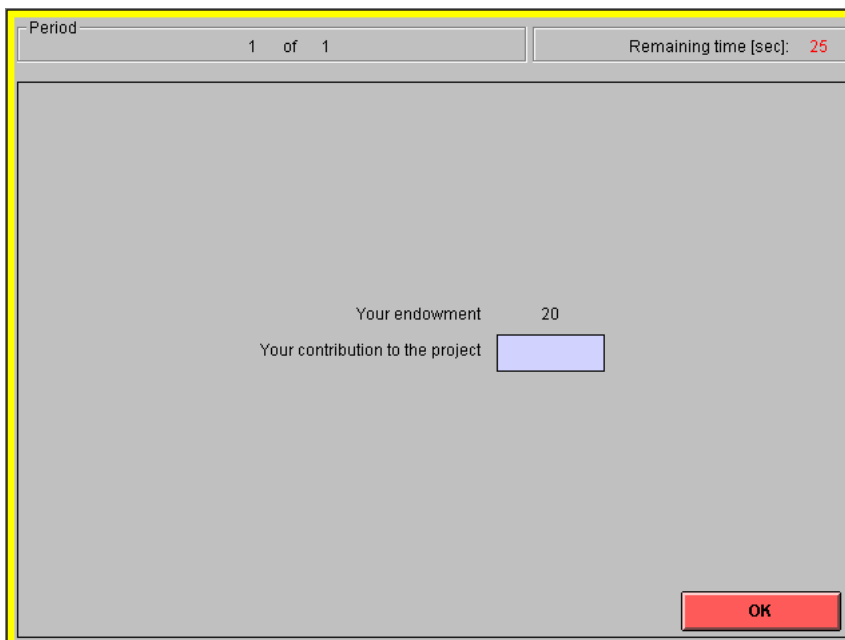
z-Leaves connect with z-Tree, you see the names of the computers appear in the first column of the clients' table.



Figure 2.10. Clients' Table with two clients connected

As soon as enough clients are connected, you can start the treatment. To do this, the treatment window must be in front and then you can choose Run → Start Treatment. When you have started the treatment, the screen shown in Figure 2.11, "Client screen of the public goods treatment" appears on the computers where z-Leaf runs.



Figure 2.11. Client screen of the public goods treatment

This is the active screen of the contribution entry stage. It consists of two boxes. The *header box* at the top shows the period number as well as the time remaining. Because it is contained in the active screen of the background, it is shown in the active screen of each stage. In the *standard box* we see the items and the button we defined in the stage tree. We can now enter a value between 0 and 20 into the field next to the text "Your contribution to the project" and click the OK button. If we enter an illegal value such as 25, -7 or 2.2, a message informs us that such an entry cannot be made.

In z-Tree, in the "Clients' Table", you can observe the state the subjects are in. For each stage there are two states: The active state, indicated with stars, corresponds to the active screen and the wait state, indicated with a dash, corresponds to the waiting screen. This enables you to check whether there are still subjects who have to make their entries. When subjects make entries, you can observe their decisions in the so-called *subjects Table*. This table can be opened from the Run menu. Figure 2.12, "Clients' table

with subjects in different states" shows the situation, where subject "first" has made her entry and the subject "second" has not yet done it.



Figure 2.12. Clients' table with subjects in different states

After all subjects made their entries, their profits are calculated and displayed. Because in the profit display stage no entry has to be made, the stage ends even if the subjects do not press the continue button. When the stage ends the subjects enter the waiting screen of this stage. They stay in this screen until something new happens, e.g. a new treatment is started. When you are testing a treatment you start the treatment, check what was going wrong, correct the errors and start the treatment again. Note that while a treatment is running, it cannot be modified.

You can stop the treatment or skip a stage during your testing (for instance, if you forgot to put OK buttons in any/all of the stages). To do so, bring the client's table to the foreground and highlight the states that you want do skip. In Figure 2.12, "Clients' table with subjects in different states", you could select "*** Contribution Entry ***" by clicking on the corresponding field or select the whole state column by clicking into the small rectangular area on top of the state column. Then, select Run → Leave Stage (see Figure 2.13, "How to skip stages for testing purposes"). If your treatment has multiple periods which you want to skip as well, you can select Run → Stop after this period before leaving the stages.
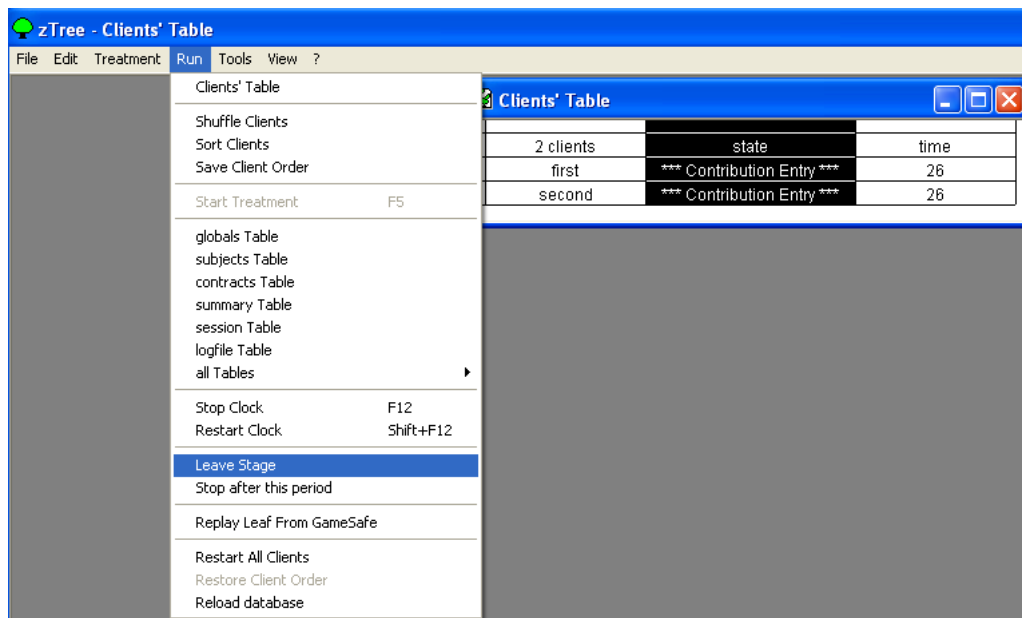


Figure 2.13. How to skip stages for testing purposes

If a stage has an input field, you will get a warning message as shown in Figure 2.14, "The warning message when leaving a stage where subjects should make input". This is a safety mechanism for real experiments, so that you do not skip stages in which you wish to gather important subject data. In

general, the leave stage option should *not* be used during a real experiment session, but only for program-testing purposes.
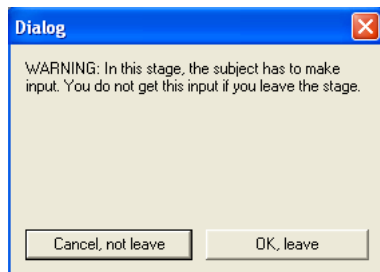


Figure 2.14. The warning message when leaving a stage where subjects should make input

Checking a treatment means that you check whether everything is calculated as you intended and that the screens look fine. If you find an error, you have to correct it (or at least try to correct it) and test the treatment again. How do you correct errors? The easiest way to correct a program is to double click the stage tree element of the program. Then you can edit, i.e., modify it. You can *modify* any stage tree element by double clicking it or by selecting it and then choosing the menu command Treatment → Info…. You can also *move* a stage tree element by selecting it and dragging it to a new place. If the element cannot be moved to this new position, the program will beep. You can *remove* stage tree elements by selecting them and choosing Edit → Cut. Finally, you can *copy* and *paste* an element: You select the element to copy and choose Edit → Copy. Then, you select the place where you want the element to be copied to and choose Edit → Paste.

We are now at the end of the guided tour. You can now exit the z-Leaves with the **Alt+F4** key combination and quit z-Tree with the menu command File → Quit. All data has been saved automatically. In Section 5.13, "Data analysis", we will show how to prepare the data for data analysis. When you quit z-Tree, a dialog with the following warning will appear: "The session is not finished with the writing of a payment file. Do you nevertheless want to quit?". You do not have to worry about this message. This message does not mean that data has not been saved it only means that you did not create a so-called *payment file* which is relevant when you conduct a session with real subjects. This will be explained in Chapter 4, *Questionnaires*.

In the next chapter, we will explain in detail how to program treatments.

## 2.7 Using another language than German

The default language is German, but you can easily change it. The language settings in ztree are relevant for the texts in *new* treatments. You can change the language in ztree in the Treatment menu or by using a command line option for ztree.

The language settings of zleaf are relevant for the error messages. You can change the language in zleaf by using a command line option. The easiest way for testing is to integrate it into the command file to start zleaf.

```
start zleaf.exe /name first /language english
start zleaf.exe /name second /language english
```

# 3 Definition of Treatments

As explained in the introduction, a *treatment* is a part of a session that is stored in a file. In the previous chapter we gave a guided tour through the process of constructing a simple treatment. In this chapter, we explain in detail how to build treatments. In the following sections, we show how increasingly complex experiments are implemented in z-Tree. Each section first contains a theoretical introduction followed by examples that illustrate the concepts presented.

## 3.1 Simple experiments without interaction

In this section, we present the most essential features of z-Tree. We first explain how data is stored in z-Tree and how data is modified by programs. Then, we show how information is displayed to the subjects and how subjects' input is processed. Finally, we give a first overview of how the course of action in a treatment is organized.

### 3.1.1 Data structure and simple programming

Information on the state of a session is stored in a *database*. This database consists of *tables*. The lines of a table are called *records*, the columns *variables*. Each variable has a name that identifies it. The individual entries in the table are called *cells*. By default, cells contain numbers. You can also store text and other kind of data into variables but all data in a column must be of the same type. We will present the different data types in Section 3.9, "Free form communication and strings". In Figure 3.1, "The most important tables in the z-Tree database", you see a screen shot of two tables, the globals table and the subjects table, as they appear in an empty treatment. The first row shows the names of the variables. In this example, the globals table contains one record and the subjects table contains 3 records. When a treatment is running, you can view the tables. You just have to choose the table in the Run menu.

| globals table | | |
|---|---|---|
| Period | NumPeriods | RepeatTreatment |
| 1 | 1 | 0 |

| subjects table | | | | | |
|---|---|---|---|---|---|
| Period | Subject | Group | Profit | TotalProfit | Participate |
| 1 | 1 | 1 | 0 | 0 | 1 |
| 1 | 2 | 1 | 0 | 0 | 1 |
| 1 | 3 | 1 | 0 | 0 | 1 |

Figure 3.1. The most important tables in the z-Tree database

The name of a variable can be any word. To be precise, a variable can be composed of letters, numbers, and the underscore character "_". The first character of a variable must be a letter. Diacritical marks, blanks and punctuation marks are not allowed. z-Tree distinguishes between uppercase and lowercase letters, so `hallo`, `Hallo` and `hAllo` are three different variables. You can give long names to variables. This makes programs easier to understand. If you want to give a variable a name made up of several words, you can separate the words with underscores or you can begin each word with a capital letter.

Examples of allowed names of variables:

```
A
contribution23
v_2_13
Buyers_offer
BuyersOffer
```

Not allowed as names of variables:

```
12a           // starts with a number
v 2 13        // contains spaces
v.1.0         // contains dots
Buyer'sOffer  // contains apostrophe
Hélène        // contains diacritical marks
```

Some variables are predefined, i.e., they appear in every treatment. Other variables are specific to a treatment – they are defined in *programs*. Programs are placed at the beginning of stages or into the Background between the table definitions and the Active screen. Programs are always defined for a particular table – in other words, they are always executed in a particular table. This table is declared in the program dialog as shown in Figure 3.2, "In the program dialog you have to specify which table the program will run in". In this dialog, the dropdown menu contains all tables available in the treatment.
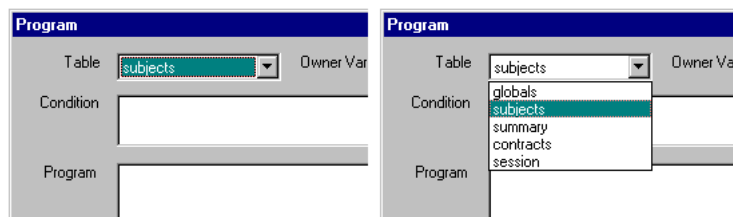


Figure 3.2. In the program dialog you have to specify which table the program will run in

The easiest table is the *globals table* because it contains exactly one record. Hence, the variable also determines the cell. So, we can talk about the value of a globals variable. For the moment, think of a program in the globals table.

The most important element in a program is the *assignment statement*. It does a calculation and assigns the result of this calculation to a variable. The syntax of the assignment is the following:

```
Name_of_variable = expression;
```

The expression on the right side of the equation is calculated and assigned to the variable on the left side. Note that each assignment statement is concluded with a semi-colon. If this variable already has a value, this value is overridden. If the variable does not yet exist, then it is created by this statement. All basic kinds of calculation are permitted in an expression. As usual, multiplication and division are calculated before addition and subtraction. In all other cases you calculate from left to right. Thus 2+3*4 makes 14, not 20, and 10-5-2 makes 3, not 7. If you want to change the order of calculation, you have to use brackets. A series of functions is furthermore available, such as min, max, exp, random, and round. The reference manual includes a complete list of all operators and functions that an expression can contain. All variables used in an expression must have been created previously, i.e., they must be defined

in the program above or in the stage tree above. It is a good practice to define variables at the beginning of a treatment. You can also use comments to explain the purpose of each variable (see Section 3.1.2, "Comments"). This makes the program much more easily accessible - for other experimenters and for your future self.

> It makes sense to write comments on the purpose of variables. But even better, use descriptive variable names. So, better than `T1=1; // D means treatment with delegation` is `TreatmentWithDelegation =1;`.

Examples of assignment statements:

```
p = 20;
Q = q1 + q2;
Profit = Endowment - Contribution + EfficiencyFactor * SumC / N;
Cost = exp( Effort / k );
```

We have seen that the globals table contains exactly one record. Other tables may contain more records. If a program is executed in such a table, there is always one record for which the program is executed. This record is called the *current record*. This allows us to omit the index for the record number. Consider, for instance, the predefined *subjects table*. This table contains one record per subject. Programs in the subjects table are executed separately for each subject, i.e., separately for each record. So, whenever a program is running, the table and the record is fixed and therefore, within a program, the cell is determined by the variable. Let us make this clear with an example. Assume that a treatment is defined for three subjects. The value of the variable g is 5, 12 and 7. Consider now the following program:

```
M = 20;
x = M - g;
```

The program is executed for each row of the subjects table. First, it is executed for the first row. The variables M and x are defined and get their values for the first row:

| g | M | x |
|---|----|----|
| 5 | 20 | 15 |
| 12 | | |
| 7 | | |

Now, the program is executed for the second row. Therefore, M and x get their values for the second row:

| g | M | x |
|---|----|----|
| 5 | 20 | 15 |
| 12 | 20 | 8 |
| 7 | | |

Finally, when the program is executed for the third row, M and x get their values for the third row:

| g | M | x |
|---|---|---|
| 5 | 20 | 15 |
| 12 | 20 | 8 |
| 7 | 20 | 13 |

In this example, the empty spaces signify undefined values. One should not use them. After a program is executed for *all* records of a table (which is generally the case), there will be no undefined cells left. It's important to note that the calculation is conducted record by record – not statement by statement. So, M is not known for all subjects when we are calculating x for the two first subjects.

The subjects table is the most important table since it contains all the data which belongs to the subjects. In many experiments, it is the only table you have to work with. Below, we will present applications where it is more convenient or even necessary to use other tables. For the moment it is sufficient if you can work with the subjects table.

### 3.1.2 Comments

In order to be able to easily understand a program days and weeks after you have worked on it, you insert comments. All text between /* and */, as well as between // and the end of the line, is a comment and of no consequence for the actual running of the program, i.e., when the program runs, this text is simply ignored. We will use comments in our example to explain the reasons why each statement has a particular form.

Examples:

```
a=1; // initialize
b = sum( /* cos(x*x+) */ a );
// there is an error in the expression in the second line;
// therefore we put questionable parts into a comment to
// localize the error
```

Comments with /* and */ may not be nested: After /*, the first occurrence of */ terminates the comment. The following program is therefore illegal:

```
/* discard the following lines by putting them into a comment
a /* first variable */ =1;
b = 2;
*/
```

You may also use comments to locate errors. To find an incorrect statement, you can turn doubtful passages into comments until no error message appears anymore. After you correct your errors, you can remove the comments.

### 3.1.3 Simple course of action

A treatment consists of a number of *periods*. This number is fixed when you run a treatment. In every period, a number of *stages* are gone through. Each stage contains two screens that are shown to the subjects who are in that stage. First, the *active screen* is shown. In this screen, subjects can make entries

or view information. When the active screen of a stage is shown, we also say that the subject is in the *active state* of that stage. When data has been entered or when time has run out, subjects move to the *waiting state* of that stage and the second screen that belongs to this stage appears, which is called the *waitingscreen*. From the waiting state, subjects can enter the next stage. Normally, subjects progress to the next stage when all subjects have reached the waiting state. At the beginning of each stage, calculations are carried out. These calculations are defined in *programs*. When a subject enters a stage, the calculations for this subject are carried out. For programs in the subjects table this means that the program is executed for the record of the subject who enters the stage.

The default procedure for how subjects proceed from stage to stage is as follows. All subjects enter the first stage of the first period when the treatment is started. If no input has to be made, the stage is left when the button is pressed or when the timeout has expired. In a stage in which input has to be made, the stage is only left when the button is pressed. Otherwise, it could happen that an inattentive subject loses the opportunity to make the input. Leaving the stage means that subjects move on to the waiting-screen of the stage. They can enter the next stage as soon as *all* subjects have left the stage. This means that the last subject(s) will not see the waitingscreen but instead immediately enter the next stage. A period ends when all subjects are in the waiting state of the last stage. If it was not the last period, the next period starts. After the last period, the last waitingscreen remains on the subject screen and the subjects' state changes to "Ready" meaning that a new treatment can now be started.

The Background element in the stage tree contains the information that is not specific to a particular stage. In the background dialog, you set for instance the number of periods and the number of subjects. Then, the Background contains the list of tables that are used in the treatment. After the tables, the Background may contain programs. They are executed at the beginning of each period. The elements that are contained in the Active screen of the Background are inserted into the active screen of each stage (see also Section Section 3.4, "Screen layout" on layout). For instance, you can display a header that shows the current period and the remaining time in every stage. By inserting it in the Active screen of the Background it is not necessary to include this header in every stage. The elements that are contained in the Waitingscreen of the Background are inserted into the waitingscreen of each stage. Very often the Waitingscreen contains only a message such as "Please wait until the experiment continues". In this case, this message can be defined in the Background and the waitingscreen in each stage can be left empty. Figure 3.3, "Course of action" shows the relationship between the elements in the Background and the elements in the stages in an abstract example with two stages.
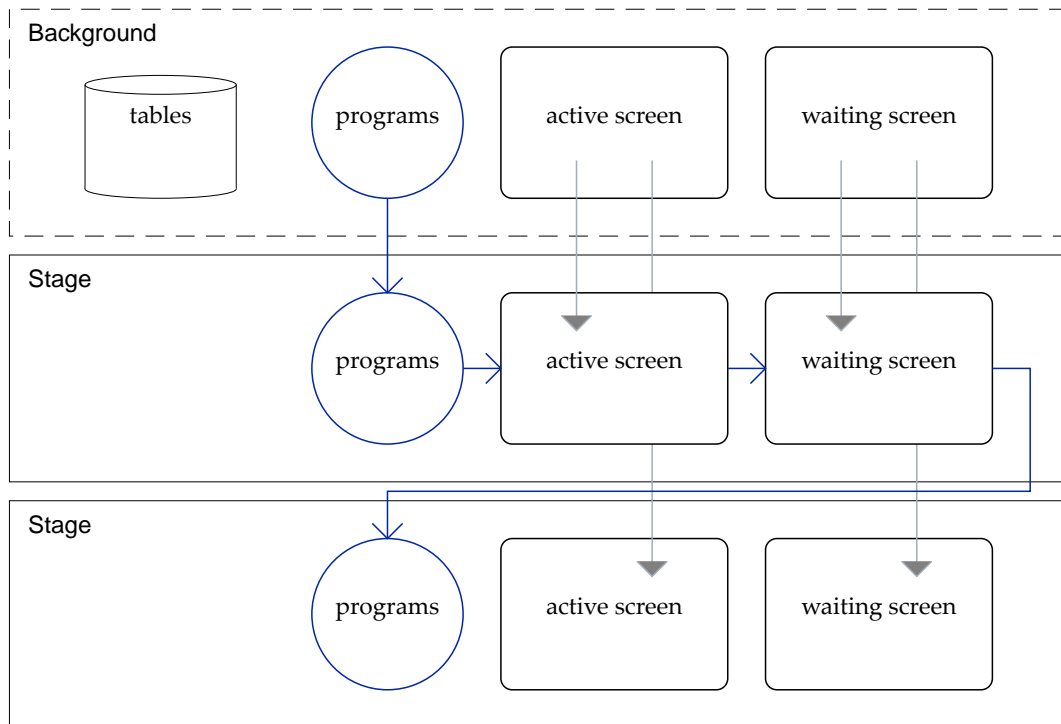
Figure 3.3. Course of action

Each stage starts with some (zero or more) programs. Then subjects are shown the active screen and the waitingscreen. The Background contains the list of tables, programs that are executed at the beginning of the period and screen elements that are inserted in every stage.

### 3.1.4 The Stage tree

The stages of a treatment are depicted in the shape of a tree diagram, called the *stage tree*. Figure 3.13, "Stage tree of the public goods experiment" above shows the stage tree of a public goods game. All elements of the treatment are arranged hierarchically in this figure: Stages contain programs and screens, screens contain boxes, and boxes contain items. In order to have an overview, you can expand and collapse the lower hierarchy levels as you wish. By double-clicking, you can view and change the parameters of the elements. Besides this, the elements may be moved and copied. You can insert new elements with menu commands. New elements are placed either after the selected element on the same level or at the first position within the selected element. This means for instance, that you have to select the preceding stage to insert a new stage – if you have selected the Waitingscreen element, you will not be able to insert a new stage because a stage cannot be placed at the same or at a lower level of a screen.

There are two "in" relations in the stage tree. A stage tree element x can contain other elements y, i.e. the elements y are placed within the element x. For instance, a program is contained in a stage. On the other hand, the properties of the element x are also within x – they can be found by double clicking the element. To make this difference more clear, we will use the "within" relation only for the elements and not for the properties, i.e. we will say that the elements y are within x. For the parameters, we do not use the "in" metaphor; we will say at most that a property of x is (defined) within the *dialog* of x. For instance, the name of the stage is defined in the stage dialog.

### 3.1.5 Data display und data input

The screen layout determines what data is displayed and what data input has to be made by the subjects. Screens consist of boxes – rectangular areas of the screen. There are different types of boxes. In this chapter, we only present the *standard box*. Standard boxes are particularly important because they can contain *items*. An item is a stage tree element that allows you to display an entry of a table. The item dialog contains the Label field where you can enter text. In the Variable field you can enter the name of a variable. If a standard box contains an item, then the value of the variable in the record of the subject will be displayed. This value is labeled with the text in the Label field. The Layout field is used to describe *how* the variable should be displayed. For instance, the number 12 can be displayed as 12 or as 12.00. In the Layout field, we write the precision with which we display the variable. For example, it is 1 if we want to display 12 and .01 if we want to display 12.00. In the standard box, items are shown in a list form: The labels are right aligned and are placed to the left of the values of the variables. By inserting empty items, i.e., items with no label and no variable, you can create vertical space between items.

If we want subjects to make an entry, we also create an item. To declare that input must be made, we check the Input checkbox. In this case, we call the item an *input item*. Other items are called *output items*. The name entered in the Variable field is again the name of a variable in the subjects table. It is not necessary that this variable is defined first since input items also *define* variables. When you check the Input checkbox, more fields appear in the item dialog. In the Minimum and Maximum fields, we declare the lower and upper bounds of the value that the subjects may enter. In the Layout field, we enter the precision of the number. If the number entered by a subject is not a multiple of the value entered in the Layout field, or if the value is not within the declared bounds, an error message appears on the subject's screen. In Figure 3.4, "Input item dialog", we show the dialog of an input item. This item defines a field on the subjects' screens. In this field the subjects must enter a number. The value of this number must be between 0 and 12 and it must be a multiple of .1. Therefore, it is possible to enter 4.8 but it is not possible to enter -2, 15 or 4.55.



Figure 3.4. Input item dialog

At the end of a standard box, we can place a Button element, so that a button appears in the standard box on the subjects' screens. If a subject presses the button, it is first checked whether all conditions for the input items are satisfied. If so, the stage is concluded. If there are input items, then there must also be a button to conclude the input.

### 3.1.6 The Variables `Profit` and `TotalProfit`

In most economic experiments, people are paid based on their decisions in the experiment. In z-Tree, the bookkeeping of the subjects' earnings is automated. You only have to make sure that at the end of a period, the predefined variable `Profit` contains the number of points (a point is the experimental currency unit) earned in that period. At the beginning of a period, `Profit` is always set to zero, so if you do not change the variable, the subjects will earn nothing. During the treatment, the profit is summed up in the predefined variable `TotalProfit`. It contains the sum of the values of the `Profit` variable in all periods in this treatment – including the current period. `Profit` and `TotalProfit` are variables in the subjects table. You can also display their values. However, you should not change the value of `TotalProfit`.

At the end of the treatment the value of `TotalProfit` is exchanged into the local currency unit (such as CHF, €, or $). The default value of the exchange rate is 1. It can be changed in the Background dialog. We will explain in chapter Chapter 5, *Conducting a Session* how the sum of all profits can be accessed to pay the subjects.

### 3.1.7 Calculation exercise

**Exercise**

The subjects have to try to calculate the sine function for a randomly determined value. They are paid according to the precision of their calculation.

**Solution**

In the stage tree in Figure 3.5, "Solution for calculation exercise" you see a solution for this experiment. There are two stages: In the input stage the estimation has to be made and in the profit display stage the payoff as well as other information is displayed.
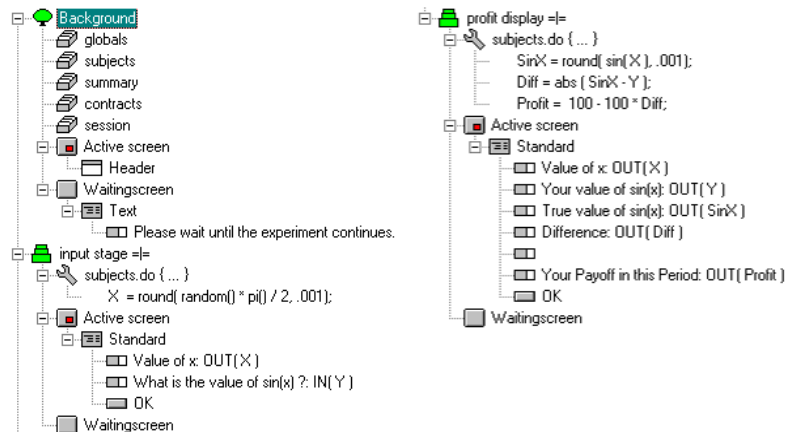
Figure 3.5. Solution for calculation exercise

The value `X` is a number between 0 and ½π with a precision of .001: The function `random()` returns a uniformly distributed number between 0 and 1, `pi()` returns 3.14159… and the `round()` function rounds the resulting number to a precision (in this case) of .001. This corresponds to a precision of three digits.

The input variable `Y` must be between 0 and 1 and we let the subjects enter a precision of at most three digits. This information has to be entered in the item dialog of the input item.

The payoff calculation is straightforward. First, we calculate the sinus function. Because the subjects can only enter a finite precision, we round the value to the same precision as that of the subjects' entries. Then, we calculate the absolute difference between the actual value and the guess entered. Finally, we calculate the profit in this period.

```
SinX = round( sin( X ), .001);
Diff = abs ( SinX - Y );
Profit = 100 - 100 * Diff;
```

In the active screen of the profit display stage, we show this information to the subjects.

### 3.2 Interactive experiments: Symmetric games

With the knowledge of the previous chapter, you should be able to program a treatment for an individual-decision-making problem. In this section, we explain how to program interactive experiments. The difference between an individual-decision-making problem and games is the fact that in games the payoff also depends on the decisions of the *other* subjects. In the terminology of z-Tree, this means that we need a feature to access cells in other rows than the row we are currently calculating in. *Table functions* serve exactly that purpose.

In an experiment, we often want more than one group to play. For instance, if we conduct a public goods experiment, we will invite 24 subjects and we will form 6 groups of 4 subjects. You can do this easily with z-Tree. There is also great flexibility in how groups can be matched. An introduction is given in this section.

### 3.2.1 Table functions

In the public goods example, we need to calculate the sum of all contributions made by all members of a group. The expressions described so far always refer only to the current record. In this example, we need to carry out calculations over the whole table. We call such calculations *table functions*. For instance, if C is the variable of the contribution, then

```
S = sum(C);
```

defines a new variable S which is the sum of the contributions of all subjects. The variable C that appears in this expression now no longer belongs to the same record as S. If *i* is the number of the current record, then the expression above mathematically means

$$S_i = \sum_j C_j$$

So, *j* loops over all records.

Of course, the argument of a table function may again be an expression. This expression is then calculated for every record of the table and these results are added in the case of the sum table function. In this way the program

```
x = sum( cos(a * b) );
```

corresponds to the mathematical expression

$$x_i = \sum_j cos(a_j b_j)$$

In every table function you can insert a condition as a first argument. This condition is checked for every record and the table function only applies to the records that satisfy this condition. Example:

```
y = average( a > 0, b );
```

Here, we calculate the average of the variable b of all the subjects who have a positive variable a.

With the find function you get the value of a cell in another record.

```
z = find( c == 12, d + e );
```

Here a record is sought for which the variable c has the value 12. For the first record from the top that satisfies this condition, the value of d + e is calculated and assigned to the variable z (z in the *current* record).

### 3.2.2 How programs are evaluated

Since each statement in a program is evaluated for all the records in a table, it is important to know that programs are executed record-by-record and not statement-by-statement. As long as there is no interac-

tion, this difference does not matter. However, as soon as table functions are used, one has to be careful. To show the problem, consider the following example. There is a good. Each subject has an endowment of the good and each unit of the good has a value for the subjects, and that value differs between the subjects. The data is stored in the subjects table in the variables `Endowment` and `ValuePerUnit`. The following program is supposed to calculate the total value of the good:

```
// WARNING: flawed program
Value = Endowment * ValuePerUnit;
TotalValue = sum( Value );
```

Consider the following table. `Endowment` and `ValuePerUnit` are set, `Value` and `TotalValue` are not yet defined, but when accessed the cells already contain data. What the values return is undefined but often they contain zero.

| Endowment | ValuePerUnit | Value | TotalValue |
|-----------|--------------|-------|------------|
| 2 | 3 | – | – |
| 4 | 5 | – | – |

First, `Value` is calculated for the first record, for a value of 6. Then, `TotalValue` is calculated for the first record. It returns the sum of 6 and an undefined value – probably 0.

| Endowment | ValuePerUnit | Value | TotalValue |
|-----------|--------------|-------|------------|
| 2 | 3 | 6 | 6 |
| 4 | 5 | – | – |

Then, `Value` is calculated for the second record, for a value of 20. Finally, `TotalValue` is calculated for the second record. It returns the correct sum of 26.

| Endowment | ValuePerUnit | Value | TotalValue |
|-----------|--------------|-------|------------|
| 2 | 3 | 6 | 6 |
| 4 | 5 | 20 | 26 |

This example shows that z-Tree processes the program row-by-row, and not column-by-column. However, to get a correct value for `TotalValue`, we need to do the calculation column-by-column. We can achieve this by separating the two calculations into two separate programs as the part of a stage tree in Figure 3.6, "Column-by-column calculation" shows.



```
subjects.do { ... }
      Value = Endowment * ValuePerUnit;
subjects.do { ... }
      TotalValue = sum( Value );
```
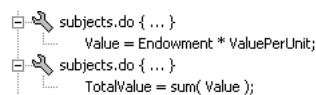
Figure 3.6. Column-by-column calculation

The first program is executed for all records, and then the second program is executed for all the records. This corresponds exactly to a column-by-column processing, as we need it in this example. There is a general rule to avoid this error. Whenever you use a table function, all variables used in this table function have to be defined in the stage tree above the program that contains the table function.

> In this particular example there is also another solution which can be implemented within one program.

```
TotalValue = sum( Endowment * ValuePerUnit );
```

### 3.2.3 Table functions in other tables

Table functions can also be evaluated in other tables. In order to do this you simply put the name of the table followed by a dot before the table function. For example, if you wish to calculate the average profit of all subjects and put it into the globals table, you write (in a program for the globals table):

```
AvProfit = subjects.average( Profit );
```

You can always use this notation, i.e. you could also use this prefix notation in a program for the subjects table. If you do not explicitly specify the table, then the current table is used. This means, that when you execute the program `AvX = average( X );` in the globals table, the average will be calculated in the globals table and the variable X refers to a variable in the globals table. For nested expressions, the current table is the table in which a variable at the place of the table function would be executed. Consider the following program of the globals table

```
Y = globals.product( A + subjects.average( sum( B ) - C ));
```

In this program, the `product` table function does not need the globals prefix and the `sum` table function is evaluated in the subjects table – as the variable C.

### 3.2.4 The scope operator

Imagine, you have three tables `ta`, `tb`, and `tc`. In these tables, we have the variables A and X in a, B on b and C in c. The meaning of the following statement is clear.

```
X = A + b.sum( B - c.product( C ) );
```

What about the following statement?

```
X = A + b.sum( A * B - c.product( A + B + C ) );
```

It is intuitive that the variable A in the sum is still from the record in which we define X. This is correct, but this expression is a bit risky. If we define a variable A in the table tb, then this expression becomes ambiguous, and z-Tree actually assumes that the variable A in the sum belongs to table tb.

in order to avoid this problem, the variable A in the sum must be preceded by a colon. This colon is called the *scope operator*. It means that we do not want a variable A that is in table tb, but the same variable A that is outside of the sum. We see also that in the nested function product, we have to use two times the scope operator - in order to "jump twice outside of the table function".

```
X = A + tb.sum( :A * B - tc.product( ::A + :B + C ) );
```

We will use the scope operator in many places, so try to understand its use and practice it. We now show an example, in which the scope operator is unavoidable. Let us suppose you want to calculate the expression

$$X_i = \sum_j cos(A_i B_j)$$

Now `A` and `B` are both in the same table. This is the same expression as in Section 3.2.1, "Table functions" except for the fact that we wish to use, in every summand of the sum, the variable `A` of the record in which the cell $X_i$ lies, and not the `A` of the record of $B_j$. In order to express this, the variable `A` must be preceded by the *scope operator*.

```
X = sum( cos( :A * B ) );
```

Let us look at a table with three records in which the variable `A` has the values 2, 4 and 8 and the variable `B` has the values 5, 12 and 7. After the execution of the following program, the table contains the values as shown in the table below.

```
C = sum( A * B);
D = sum( :A * B);
E = sum( :A * :B);
```

| A | B | C = sum( A * B ); | D = sum( :A * B ); | E = sum( :A * :B ); |
|---|---|---|---|---|
| 2 | 5 | 10 + 48 + 56 = 114 | 10 + 24 + 14= 48 | 10 + 10 + 10 = 30 |
| 4 | 12 | 10 + 48 + 56 = 114 | 20 + 48 + 28 = 96 | 48 + 48 + 48 = 144 |
| 8 | 7 | 10 + 48 + 56 = 114 | 40 + 96 + 56 = 192 | 56 + 56 + 56 = 168 |

Another intuition for the scope operator can be given by considering an example in which a table function is calculated in another table. So let us consider a variable `V` that appears in the tables `ta` and `tb`. Let us consider the expression `tb.sum(V)` used in a program for table `ta`. In this expression, `V` is the variable `V` in table `tb`. If we want to access the variable `V` in table `ta`, we have to use the scope operator. Because `ta` and `tb` are different tables, it becomes more clear what the current record is. Outside of the table function, it is the current record in `ta`, inside of the `sum` function, it is the current record when we calculate the sum. This record is in `tb`. Nevertheless, within the calculation of the sum as well, the current record in `ta` can be accessed with the scope operator.

When we execute a table function, we calculate an expression for every record of that table. With every step another record becomes the current record. With the scope operator we may go back to the variables of the "old" current record, the record that lies *outside* the table function. In this respect, the scope operator gives us access to a wider scope of cells.

When a table function is carried out within another table function, this results in an expression of this table function with *three* records that are accessible. Let us consider the following expression where all three tables `ta`, `tb` and `tc` contain a variable `V`. In the expression of the product, `V` is the variable in the current record of table `tc`, the 'scoped' variable `:V` is the variable in table `tb` and only by doubling the scope operator, `::V`, do we reach the cell in the current record of table `ta`. The line underneath the expression specifies which `V` is being used at a particular place.

```
X = V + tb.sum ( V * :V - tc.product ( V - :V - ::V ) )
//  ta          tb  ta                  tc  tb   ta
```

In this example the variable V occurs in all tables. However, it may happen that a variable only occurs in one table, as in the example at the beginning of this section. In this case you may omit the scope operator. Let us suppose that the variable A occurs only in the table ta, B only in the table tb and C only in tc. Then the following expressions are equivalent:

```
X = A + tb.sum ( B * :A - tc.product( C - :B - ::A ) )
X = A + tb.sum ( B *  A - tc.product( C -  B -   A ) )
```

Note, however, that not using the scope operator can be dangerous, as mentioned above. If at some time a variable A had been defined in table tc, then the product of the second expression no longer goes back to the variable A in table ta but to the A in table tc. Note also that if the prefix tc is omitted in front of the product table function, the product is taken over the records in table tb and if the prefix tb is omitted in front of the sum table function, the sum is taken over the records in table tb.

**Example**

Consider the following two tables

| A | V |
|---|---|
| 1 | 3 |
| 4 | 6 |

| B | V |
|---|----|
| 7 | 8 |
| 10 | 11 |
| 13 | 14 |

Assume the following program is calculated in table ta. Which expressions are correct and what are their results?

```
X1= sum(V);
X2= tb.sum(V);
X3= tb.sum(:V);
X4= sum(:V);
X5= tb.sum(:V);
X6= sum( product (V) );
X7= sum( product (:V) );
X8= sum( product (::V) );
X9= tb.sum( product (V) );
Y1= sum(A);
Y2= tb.sum(A);
Y3= tb.sum(:A);
Y4= sum(:A);
Y5= tb.sum(:A);
Y6= sum( product (A) );
Y7= sum( product (:A) );
Y8= sum( product (::A) );
Y9= tb.sum( product (A) );
Z1= sum(B);
Z2= tb.sum(B);
Z3= tb.sum(:B);
Z4= sum(:B);
Z5= tb.sum(:B);
Z6= sum( product (B) );
Z7= sum( product (:B) );
Z8= sum( product (::B) );
Z9= tb.sum( product (B) );
```

Solution: $Z1$ and $Z3$ to $Z8$ are incorrect. The result of the other expressions equal:

| A | C | V | X1 | X2 | X3 | X4 | X5 | X6 | X7 | X8 | X9 | Y1 | Y2 | Y3 | Y4 | Y5 | Y6 | Y7 | Y8 | Y9 | Z2 | Z9 |
|---|---|---|----|----|----|----|----|----|----|----|------|----|----|----|----|----|----|----|----|-----|----|------|
| 1 | 2 | 3 | 9 | 33 | 9 | 6 | 9 | 36 | 45 | 18 | 3696 | 5 | 3 | 3 | 2 | 3 | 8 | 17 | 2 | 3 | 30 | 2730 |
| 4 | 5 | 6 | 9 | 33 | 18 | 12 | 18 | 36 | 45 | 72 | 3696 | 5 | 12 | 12 | 8 | 12 | 8 | 17 | 32 | 192 | 30 | 2730 |

### 3.2.5 The do statement

Table functions permit read access in other tables. There are situations in which this is not sufficient; there are situations in which we want to change a variable in another record or table. In Section 3.6, "Continuous Auction Markets", we show some applications of this feature. Here we present the logic and give a first application. With the do statement calculations can be carried out in all records of a table.

```
do { statements }
```

the program `statements` is executed for all records in the current table.

As in the table function, it is possible to precede the do-statement with a table name. By doing this, calculations are carried out in the table in question. As in the table functions, you may also use the scope operator here. Example: Assume the following line is executed in the globals table.

```
subjects.do {
    Money = :InitialMoney;
    :LastSubjectDone = Subject;
}
```

It sets, in all records of the subjects table, the variable `Money` to the variable `InitialMoney` in the globals table. Furthermore, it sets the variable `LastSubjectDone` in the globals table to the value of the `Subject` variable in the subjects table. The program runs through the whole subjects table, so finally `LastSubjectDone` will contain the value of the `Subject` variable of the last subject.

### 3.2.6 Application: Restricting a table function to the members of one's own group

An important use of the scope operator consists of calculating a table function restricted to the members of the subject's own group. The variable `Group` contains an ID for the group, i.e., it is 1 for the first group, 2 for the second, etc. In general, we want to restrict interaction to groups, i.e. we want to restrict a table function to the members of the own group. Suppose that we want to calculate the sum of the variables g in the own group. If we know that at most, groups 1, 2, 3 and 4 exist, we can calculate `s` without the scope operator:

```
s = if( Group == 1, sum( Group == 1, g ),
    if( Group == 2, sum( Group == 2, g ),
    if( Group == 3, sum( Group == 3, g ),
                    sum( Group == 4, g ))));
```

This expression is complex, susceptible to error, and not general. In particular, if the number of groups is higher than 4, this expression is wrong.

With the scope operator, this kind of calculation can be simplified. The following expression calculates the sum of all g in the subject's own group.

```
s = sum( Group == :Group, g );
```

We can read this expression as follows: "(My) s is the sum over the g's of those subjects whose `Group` is equal to my `Group`.". In many contexts it is correct to translate the scope operator into "my" – as we did it here. However, the correct intuition is that the scope operator refers to the "my" in front of the s. The scope operator refers to the record that contains the s we are calculating. We need the scope operator where we want to access cells in the record belonging to the s we are calculating and not cells in the record belonging to the g in the table function.

As performing table functions on one's own group is something very common, the function `same` was specially introduced for such calculations. The expression `same(x)` is an abbreviation of `x == :x`. The expression above may therefore be written in the following way:

```
s = sum( same( Group ), g );
```

Thanks to this operation, the scope operator becomes invisible. It is not necessary to understand the scope operator in detail in order to understand a program intuitively. However, if you want to write your own programs, a deeper understanding of the scope operator is crucial.

### 3.2.7 Basic group matching

With the menu Treatment → Matching groups can be set up in partner or stranger designs. Partner matching is fixed matching. The first players constitute group 1, the next players group 2 and so on. The stranger matching is a random matching, i.e., in every period, the group is determined by the computer's random generator. The commands in this menu set the Group variable in the specified way. So, if you change the number of subjects or the number of groups, you have to reapply the command! More flexible matchings can be entered in the parameter table that will be explained in Section 3.3.3, "The parameter table". In addition, you can specify the matching in a program, which is explained in Section 3.3.10, "Programming group matching".

### 3.2.8 Summary statistics

The tables mentioned so far are reinitialized after each period (of course after being stored to the disk). So, for the experimenter in the lab, they disappear from the screen. To keep an overview of the course of the experiment, you can use the *summary table*. The summary table contains one record per period but the table is not reinitialized at the end of a period. It is not initialized until the *treatment* ends.

If you run a program for the summary table, the program is only executed for the record of the current period.[1] Look at this example:

```
AvProfit = subjects.average( Profit );
```

If you conduct the above program for the summary table, then in each period the variable AvProfit is calculated and in the summary table we can view at the average profits made in the treatment.

### 3.2.9 Example: Programming the public goods game

In this section we go through the programming of a simple public goods treatment. We will present the programming on an intuitive level and go into detail in the next chapter. In this public goods treatment, the subjects are matched into groups of four subjects. They decide how many out of 20 points they want to contribute to a public good, called the project. The subjects' profit is made up of two parts: The first part is the amount they keep: 20 minus their contribution. The second part is the income from the public good: All contributions in a group are summed up, multiplied by 1.6 and distributed among all subjects in the group.

Let us now start programming this treatment. When we start z-Tree, a window containing an untitled, empty treatment is also opened as you can see in the figure above. The treatment is represented by a tree structure called the *stage tree*: A treatment is constructed as a sequence of *stages*. Before we start constructing the stages, we do some preparation. First, we set up some parameters for the treatment. In the *background*, we enter the number of subjects, the number of groups, the number of periods and how points earned are translated into the local currency. We open this dialog either by double clicking

---

[1]However, if you apply a table function, the program runs through all records. If you want to restrict the program to the current period, you have to add the condition same(Period)

the background element in the stage tree or by first selecting this line and then choosing the menu command Treatment → Info…. Figure 3.7, "General parameters dialog" shows the dialog that appears.
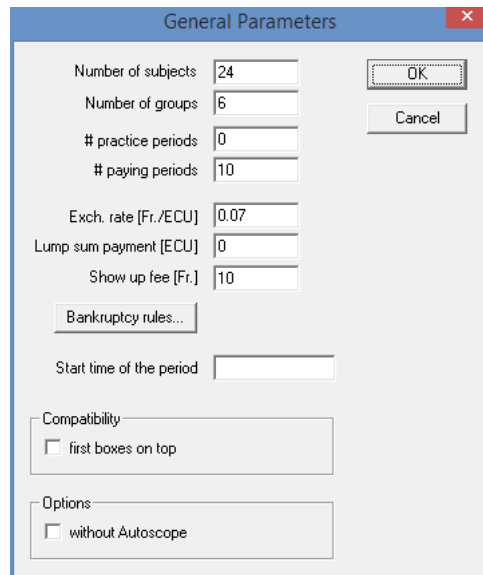


Figure 3.7. General parameters dialog

We set the parameters as follows. The number of subjects equals 24. Because we want to define groups of 4, we will have 6 groups. We have 10 repetitions and therefore we set the number of paying periods (# paying periods) to 10. There are no trial periods and therefore we do not change the zero in the field # paying periods. We set the show up fee to 10. This is the amount of money that is given to the subjects just when they show up. Thus, it is only implemented in the first treatment. In the following treatments, it is ignored. It is defined in the local currency unit (CHF in Zurich). The exchange rate defines the value of an internal point (experimental currency unit) in the local currency unit. In our example 100 points are exchanged into 7 CHF.

The parameters specific to the treatment – as the endowment 20 and the efficiency factor 1.6 – are defined in a program. To insert a program, select the stage tree element just above the Active screen in the stage tree element Background. Initially, it is the element called "session". Then choose Treatment → New Program…. In Figure 3.8, "Programs are entered in the program dialog" you find the dialog that appears.

The actual program is entered into the field Program. Enter the following code into this field:

```
EfficiencyFactor = 1.6; // return from the PG per invested point
Endowment = 20;
```

This program defines two variables. When the program is executed, the variable EfficiencyFactor is filled with the value 1.6 and the variable Endowment is filled with the value 20. It is a good practice to define all parameters at the beginning of the treatment, i.e. in the Background. This makes the treatment easier to understand and it allows making changes in the parameters at one single place. The text after the two slashes (//) contains a comment. Comments are not interpreted, i.e. a comment does not change a program's consequences. The comment is addressed to the programmer. It should make the program more comprehensible – for the programmer who looks at the program at a later time. Comments are

very valuable. It is a good practice to put a comment into the first program of a treatment which explains the purpose of the treatment.
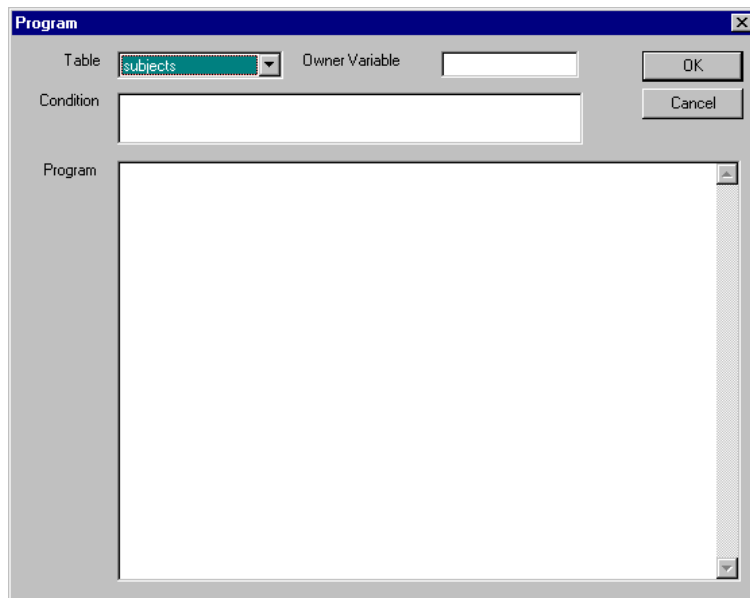


Figure 3.8. Programs are entered in the program dialog

Now, we add *stages* to the treatment. Stages are steps in a treatment. They correspond to the "essential" screens, which are presented to the subjects. In the public goods experiment there are two essential screen. On the first screen the subjects enter how much they contribute and on the second screen they get to know the other subjects' decisions and their income in this period.

To add the first stage, we select the stage tree element Background and choose Treatment → New Stage…. A dialog opens. In this dialog, we can choose some options. In our treatment, we do not have to change the default options. We do only change the name of the stage into `Contribution Entry` and click the OK button. We observe that this stage (as any stage) contains two elements (see Figure 3.9, "A stage with the active screen and the waiting screen as shown in the stage tree"): The Active screen and the Waiting screen. The *active screen* represents the "essential" screen. On this screen a subject gets information and enters the decisions. The *waiting screen* is shown when the subject has concluded the stage. It is shown until the subject can continue.
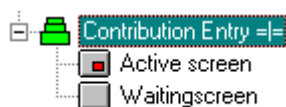


Figure 3.9. A stage with the active screen and the waiting screen as shown in the stage tree

Now, we define the Active screen of the Contribution Entry stage. A screen consists of *boxes*: rectangular parts of the screen. There are different kinds of boxes which all can be created from the hierarchical menu Treatment → New box. The most common box is the *standard box*. To create a box, we select the Active screen element in the Contribution Entry stage and choose Treatment → New box → New Standard box…. In the dialog that appears, all options are already set properly. So, we conclude the dialog with a click on the OK button. Into this box we place *items*. An item is the representation of a variable. First,

we show the endowment variable Endowment. We select the Standard box and choose Treatment →
New item…. We put `Your endowment` into the field Label, `Endowment` into the field Variable and
`1` into the field Layout. This means that the variable `Endowment` will be shown, labeled with the text
"Your endowment". The layout determines how the variable is displayed. If it is a number, it means
that the number has the corresponding resolution. Because the layout equals 1, the number 20 is displayed
as 20 and not as 20.0.
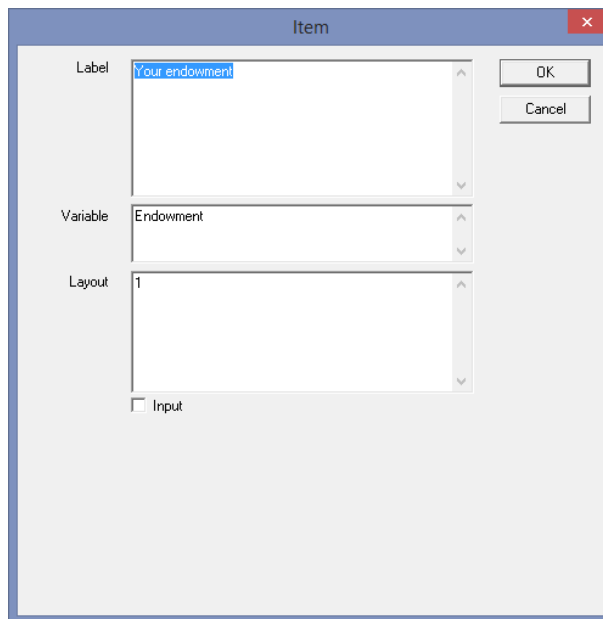


Figure 3.10. Output item dialog

The second item consists of the contribution entry. We will call this variable `Contribution`. It is an
input variable. That means not that the value of `Contribution` will be displayed, but the subject has
to enter a value and this value is assigned to the variable. Its label is `Your contribution to the
project`. It must be a multiple of `1` (entered in the field Layout) and between `0` and `Endowment`
(entered in Minimum and Maximum within the dialog). These fields appear as soon as the check box Input
is checked.

Figure 3.11. Input item dialog

The contribution entry stage will be concluded by pressing a button. This button is inserted after the input item of the variable `Contribution`. We choose Treatment → New Button…. In the dialog that appears (see Figure 3.12, "Button dialog"), we can enter the name of the button that is displayed to the subjects. In a stage where input has to be made, the default OK is a good choice. The other options in this dialog will be explained later. The default options are a good choice.



Figure 3.12. Button dialog

This completes the first stage. In our treatment, there is a second stage, the profit display stage: We select the Contribution Entry stage and choose Treatment → New Stage…. In the dialog, we name this new stage `Profit Display`.

In this Profit Display stage, we display the income of the subjects. Before we can do this, we have to calculate it. We insert a program at the beginning of this stage. It is executed for every subject when he enters this stage. To insert the program, we select the Profit Display stage and choose Treatment → New Program…. In the Program field, we enter

```
SumC = sum( same( Group ), Contribution );
N = count( same( Group ) );
Profit = Endowment - Contribution + EfficiencyFactor * SumC / N;
```

In the first line we calculate the sum of the contributions in the group of the subject. `Group` is a predefined variable. It is 1 for the first group, 2 for the second group, etc. It will be explained in detail in Chapter 3, *Definition of Treatments*. In the second line we count the number of subjects in the group. Of course, in this treatment, we could also write `N=4;`. However, the formula above allows us to run this treatment with different group sizes without changing any line of program. In the last line, we calculate the payoff for the subject. The predefined variable `Profit` is used for this. This variable is special because at the end of each period, this variable is summed up. At the end of the session, you can easily generate a file, the so-called *payment file* that contains for each subject the sum of the profits made during the whole session.

Now, we define the profit display screen: First, we insert a standard box into the Active screen of the Profit Display stage. Then, we insert three items into this box. We show:

– the own contribution,

– the sum of all contributions, and

– the subject's income for the period.

We insert the items that show the subject's own contribution and the sum of all contributions as explained above. In the item that displays the subject's income, we enter `.1` into the field Layout because we want to show the variable `Profit` with one digit precision. The profit display screen also has a button. We name this button `continue` because the subjects do not have to confirm anything. The button just allows the subjects to proceed more quickly than the default timeout.

This was it! Now we are done with the treatment. Figure 3.13, "Stage tree of the public goods experiment" below shows the stage tree of our public goods treatment. It is now a good time to save the treatment. Choose File → Save As…. A normal "save as-dialog" appears. You can give the treatment an appropriate name and store it in the directory you like.
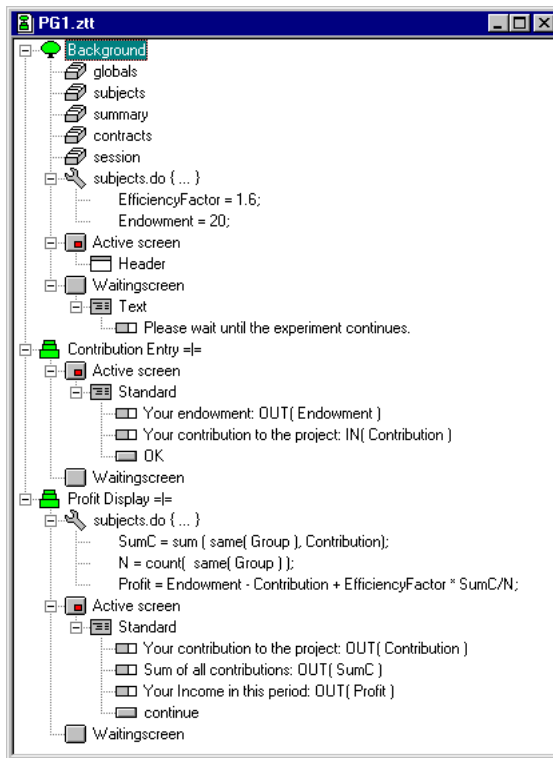
Figure 3.13. Stage tree of the public goods experiment

### 3.2.10 Example: Guessing game

**The experiment**

The subjects have to enter a number between 0 and 100. The subject who is closest to ⅔ of the average of all numbers entered wins a prize of 50 points. If there is a tie, the prize is shared equally among the winners.

### The solution
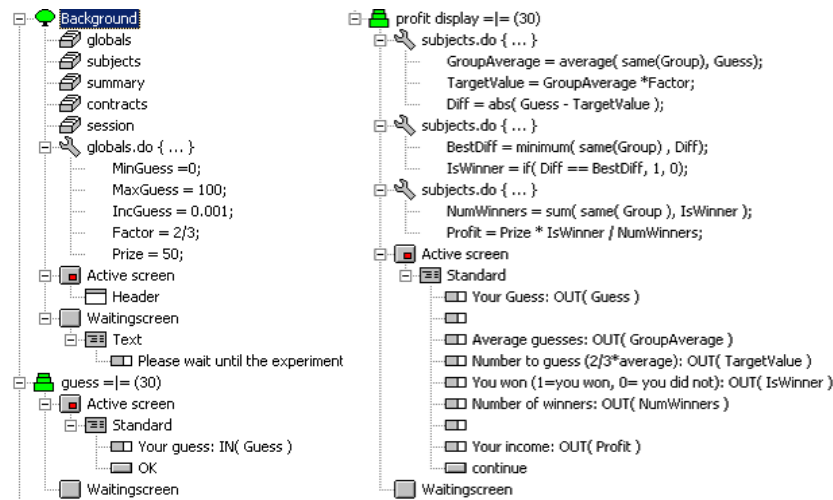


Figure 3.14. Solution for the guessing game

In the Background, we define the constants as usual. In the first stage, the subjects enter their guesses. In the second stage, we perform the calculations. Let us explain each step of the calculations.

The group average is calculated with the table function `average`.

```
GroupAverage = average( same( Group ), Guess );
```

The value to guess, called `TargetValue`, is just the product of the average and the factor of ⅔.

```
TargetValue = GroupAverage * Factor;
```

The relevant difference is the absolute value of the difference between the guess and the value to guess.

```
Diff = abs( Guess - TargetValue );
```

The smallest difference, called `BestDiff` is the minimum of all differences.

```
BestDiff = minimum( same( Group ), Diff );
```

As we have explained above, it is important that this table function is placed into a new program. Programs are executed row by row. So, if we were to place the calculation of `BestDiff` into the same program as the calculation of `Diff`, we would not calculate `BestDiff` correctly because we do not know the value of `Diff` for the records later in the table. In the current implementation, an empty value is set to zero. So, for all subjects except the last one, `BestDiff` would be calculated as zero. If we put the calculation of `BestDiff` into a new program, everything is done correctly. First, the first program is calculated for all records (i.e., for all subjects). Then, the second program is calculated for all records. At this moment, `Diff` is calculated for all records and we can apply the table function `minimum`.

In the next statement, we determine the winner(s). We put 1 into the variable `IsWinner` if a subject was closest and 0 otherwise.

```
IsWinner = if( Diff == BestDiff, 1, 0 );
```

The `if` function takes three arguments. The first argument is a condition: `Diff == BestDiff`. If this condition is satisfied, the second argument is the result of the `if` function (i.e., 1). If the condition is not satisfied, then the third argument of the `if` function is the result of the `if` function (i.e., 0).

Next, we calculate the number of winners to deal with the ties. We have to place this statement into a new program again because in the table function `sum`, we use a value that is calculated in the program above.

```
NumWinners = sum( same( Group ), IsWinner );
```

Finally, we calculate the profit. It is zero for those subjects who did not win (`IsWinner` is zero) and it is `Prize*1/NumWinners` for the winners.

```
Profit = Prize * IsWinner / NumWinners;
```

### 3.2.11 Example: Rank dependent payment

**The experiment**

The subjects have to guess the value of a mathematical function as in Section 3.1.7, "Calculation exercise", but this time subjects are paid according to their relative performance. The best player receives one point less than there are group members. The second player receives one point less than the first and so on until the last player who receives zero points. Finally, we apply a cost neutral tie rule. So, if for instance two players have rank 2 then both get a payoff according to their average rank 2.5. In the following table, we show an example. In the first line, we list an example of the rank. In the second line, we show the rank that results if we apply a tie rule that assigns the last rank in a group of players with the same rank. In the last line, we list the profit.

| Rank | 1 | 2 | 2 | 4 | 4 | 4 | 7 | 8 | 9 | 9 |
|----------|---|-----|-----|---|---|---|---|---|----|----|
| Bad Rank | 1 | 3 | 3 | 6 | 6 | 6 | 7 | 8 | 10 | 10 |
| Payment | 9 | 7.5 | 7.5 | 5 | 5 | 5 | 3 | 2 | .5 | .5 |

**The solution**

Because only the profit display stage is different from the Section 3.1.7, "Calculation exercise", we show and explain only this stage.

```
profit display =|=
  subjects.do { ... }
    SinX = round( sin( X ), .001);
    Diff = abs ( SinX - Y );
  subjects.do { ... }
    Rank = count( same( Group ) & Diff < : Diff) +1;
    BadRank = count( same( Group ) & Diff <= : Diff) ;
    N = count ( same( Group) );
    Profit = N- (Rank + BadRank)/2;
  Active screen
    Standard
      Value of x: OUT( X )
      Your value of sin(x): OUT( Y )
      True value of sin(x): OUT( SinX )
      Difference: OUT( Diff )

      Your rank: OUT( Rank )

      Your Payoff in this Period: OUT( Profit )
      OK
  Waitingscreen
```

Figure 3.15. Solution for the rank dependent payment

The second program is new in this treatment. The statements in this program cannot be placed into the previous program because we calculate table functions that use the variable `Diff` (see section above). Let us explain the program step by step.

```
Rank = count( same( Group ) & Diff < :Diff ) + 1;
```

In this line we calculate the rank. The rank corresponds to the number of players who are strictly better than I am. This is expressed with the condition `Diff < :Diff`. The `count` table function can be read as follows: We count the number of players in my group whose difference is smaller than my difference. We have to add 1 because if *no* player is better than I am, then I am first.

```
BadRank = count( same( Group ) & Diff <= :Diff);
```

In this statement, we count the number of players who are at least as good as I am. This corresponds to the rank with the 'unkind' tie rule.

```
N = count ( same( Group) );
```

This statement just counts the number of players in the group.

```
Profit = N - (Rank + BadRank) / 2;
```

The relevant rank is the average of the two ranks calculated above. The payoff depends on this average rank as calculated in this formula.

### 3.3 Symmetric, simultaneous games

In the previous chapter, all subjects had the same parameters. In this chapter, we explain how to implement different parameters for different subjects. There are essentially two ways. One way is to calculate the parameters conditionally with if functions and statements. The second way is to use the parameter table. It is strongly recommended that you stick to one of the methods. It makes it easier to read your treatments. If you feel comfortable with programming, it is perhaps easier not to use the parameter table because in this case, you have all the information at the same place. The parameter table can be helpful

if you conduct experiments with different types of players because matching and type assignment is more obvious in the parameter table than in a program in the stage tree.

### 3.3.1 Conditional execution

In z-Tree, it is possible to execute programs depending on the value of the entries in the database. This is done by using *conditions*. Conditions are expressions that do not represent a number but a *logical value*, i.e., either true (`TRUE`) or false (`FALSE`). An example of a condition is `g >= h`. This condition results in `TRUE` if and only if the variable `g` is at least as great as the variable `h`. Another example, the condition `m == n` results in `TRUE` if and only if the variables `m` and `n` are equal. Note that in contrast to the equals sign in the assignment statement, the equals sign used in conditions consists of two equals signs.

Conditions are important because they can be used in `if` functions and `if` statements. With

```
if( c, x, y )
```

a *conditional calculation* can be carried out. Here, `x` and `y` are 'normal' expressions and `c` is a condition. If `c` evaluates to `TRUE`, the if expression returns the value of `x`, otherwise it returns the value of `y`. Because `x` and `y` can be any expression, they can also contain `if` functions. In this case we say that these `if` functions are nested (you can of course nest any kind of function). The following expression shows a nested `if` function that implements a profit function for a two person game in which both players can choose either 1 or 2. `Profit11`, `Profit12`, `Profit21`, and `Profit22` describe the payoff matrix for the subject. The variable `Profit21`, for instance, contains the payoff for the subject if the subject chooses 2 and the other subject chooses 1.

```
Profit = if( Decision == 1,
         if( OthersDecision == 1, Profit11, Profit12 ),
         if( OthersDecision == 1, Profit21, Profit22 ));
```

If you wish to calculate a condition and keep it for future use, you have to store it in a normal variable, i.e. in a number. In this case you use the standard 0 for `FALSE` and 1 for `TRUE`.

An alternative to the *if function* is the *if statement*. There are the following forms.

```
if( condition ) {
    statements1
}
```

or

```
if( condition ) {
    statements1
}
else {
    statements2
}
```

If the condition evaluates to TRUE, then all of the statements between the first pair of curly brackets are executed (statements1). If it evaluates to FALSE, then in the first case nothing happens and in the second case, all the statements between the pair of curly brackets after the else statement are executed (statements2). An assignment statement using an if function can be translated into an equivalent if statement. For instance,

```
z = if( c, x, y);
```

is equivalent to

```
if (c) {
    z = x;
}
else {
    z = y;
}
```

In this example, the second form is more complicated. However, it is better suited in cases where we use one condition in several expressions. For instance, this is the case when we have different types of players with completely different payoff functions.

If we have to differentiate between more than two situations, then the elsif statement can be used. It is explained in Section 3.10.1, "Conditional execution of statements".

### 3.3.2 Calculating different values for different subjects

There are variables called Period and Subject that are set by z-Tree. Period contains the number of the current period where 1 stands for the first paid period. The variable Period is defined in every table.[2] The variable Subject is defined in the subjects table. It contains the number 1 for the first record, 2 for the second and so on. So, the variable Subject allows you to identify each subject.

You can now define variables in subject or period-specific terms by using these variables:

```
if( Period== 1 & Subject ==1 ) {
    p = 11;
}
if( Period== 1 & Subject == 2 ) {
    p = 12;
}
```

In the next section we show how the parameter table is used to define subjects specific parameters.

### 3.3.3 The parameter table

For each treatment, there is, besides the stage tree, a *parameter table* where subject-specific variables may be managed . In this table, periods are shown in the rows and subjects in the columns.

---

[2]To be precise, the variable Period is defined in each table with lifetime period. Lifetime of variables is explained in Section 3.13.1, "Definition of new tables".

Figure 3.16. Parameter table dialog

The subjects are named "S 1", etc. and the periods are numbered. Trial periods are preceded by the term "Trial". In the cells of this table, programs can be entered. These programs are executed for the corresponding periods and subjects. For instance, the program in the cell "S 2" is executed for subject 2 in every period. This means that this program is executed in every period for the second row in the subjects table. In this cell, we define the *role parameters*. The program contained in cell "F" is also executed for subject 2. However, it is only executed in period 1, the first paid period. In this cell, we define the *specific parameters*. The program in the cell label with "1" (first column) is executed in the globals table. In it, we define parameters that are the same for all subjects (but differ from period to period). In this cell, we define the *period parameters*.

The programs in the parameter table can be viewed and changed with the command Treatment → Info… or by double-clicking the field in question. Whole cells can be copied either with Copy/Paste or by selecting them and then dragging them over from one field to another.

In the cells of the specific parameters, there is a small number in the upper left corner. This is the group number. This number can be modified in the dialog of the cell or by applying the matching procedures in the Treatment → Matching menu.

For every program, z-Tree checks that no undefined variables are used. Variables are defined in input items and when they get assigned a value. However, assignments in the parameter table are not considered definition. This means that variables that are only defined in the parameter table are not considered as defined in the stage tree, so you have to initialize them in a program in the Background of the stage tree. This also guarantees that these variables always have a default value – even if they are not defined in every cell in the parameter table. Values that are assigned in the Background can be overridden in the parameter table, because at the beginning of a period, the database is set up as follows:

1. Setting of standard variables as `Period`, `Subject` and `Group`.

2. Running of programs in Background.

3. Running of subject program (in current period) in the subjects table.

4. Running of role program in subjects table.

5. Running of period program in globals table.

6. Running of programs of first Stage.

**Example**

Let us suppose that there are different types of subjects, e.g., type 1 and type 2. In this case you first define a default value for the variable in the Background; e.g.

```
Type = 0;
```

(You can use an illegal value here only if you redefine the variable in *every* cell.) Then, you double-click the fields in which you wish to change the value. In this dialog, you first give this cell a name that represents the type. This makes it easier to manage the parameter table. Then you enter the program in which the value is changed. For instance:

```
Type = 1;
```

It is a good practice in programs to use names instead of numbers whenever this is possible. So, in the public good game we defined a variable `Endowment` that kept the value of 20. This makes the program easier to understand and easier to change. When we have values that distinguish different types, it is reasonable to define variables for the type options in the globals table. If we have proposers and responders in an experiment, we could define:

```
PROPOSERTYPE = 1;
RESPONDERTYPE = 2;
```

Then, you can define the type as:

```
Type = PROPOSERTYPE;
```

If types are binary, it makes sense to define variables as binary types, where 1 expresses TRUE and 0 expresses FALSE. In the case of the ultimatum game, we could use variables as `IsProposer` and/or `IsResponder`.

### 3.3.4 Group matching

The variable `Group` determines the *group matching*. This variable is initialized at the beginning of the period when the other standard variables are initialized. It is then set to the value in the upper left corner of the cell of the specific parameters. You can modify this value in the dialog or with the menus in the Treatment → Matching menu. So you can for instance define fixed (partner) or random (stranger) matchings. Whenever you apply a matching command, this command is applied to the selected area in the parameter table or – if there is nothing selected – to the whole parameter table. The matching commands directly modify the number in the upper left corner in the cell of the parameter table. They do not define a logical matching structure. So, if you change the number of subjects or periods in a treatment, you have to reapply the matching command.

You can also change the `Group` variable in programs. This allows endogenous matching or the definition of matchings that are independent of the number of subjects (see Section 3.3.10, "Programming group matching").

### 3.3.5 Importing parameters

Entering the values into the parameter table by hand is only reasonable if there are not too many variants. It is for instance useful, if there are a couple of parameter combinations which differ from period to period in an unsystematic way. If a parameter is different for all periods and subjects, it becomes impractical to double-click the field in question and to adapt the program for every cell of the parameter table. Instead you can set up a tab-separated table with the variable values in an editor and import it with the command Treatment → Import Variable Table…. This command is available if the parameter table is the active window. If you choose this command, you first have to enter the name of the variable you want to import. Assume you name it `MyVar`. Then, you choose the file where you have entered the values. Then the file is processed: Wherever there is a nonempty entry in the table, then the following line is added to the program of the corresponding specific parameter. If the value in the table equals 45, for instance, the line added will be:

```
MyVar = 45;
```

The import just wraps the text in the table with "`variable =`" and a semicolon. Knowing this, it is possible to import any program into the parameter table. The program only has to start with an assignment. However, adding a dummy statement at the beginning can easily do this.

When you have imported variables and program, it is useful if they are visible in the representation of the table. You can achieve this with the menu commands Treatment → Show Variable…, Treatment → Append Variable… and Treatment → Append Text…. The commands are explained in the reference manual.

### 3.3.6 Importing parameters into a program

If you want to import many different values, then is also possible to generate a z-Tree program in a spreadsheet program. Imagine you created the following table in a spreadsheet program:

|   | A | B | C | D |
|---|---|---|---|---|
| 1 | Subject | V1 | V2 | V3 |
| 2 | 1 | 5 | 6 | 7 |
| 3 | 2 | 8 | 9 | 10 |

You would like to generate a program like:

```
if ( Subject == 1 ) { V1=5; V2=6; V3=7; }
if ( Subject == 2 ) { V1=8; V2=9; V3=10; }
```

This can be done using formulas in the spreadsheet program. For example `V1=5;` can be generated using:

```
=B$1 & "=" & B$2 & ";"
```

### 3.3.7 Exercise: General 2x2 game

In this example, we program a battle of the sexes game. We program the example in such a way that we can use it for any two player 2x2 game.
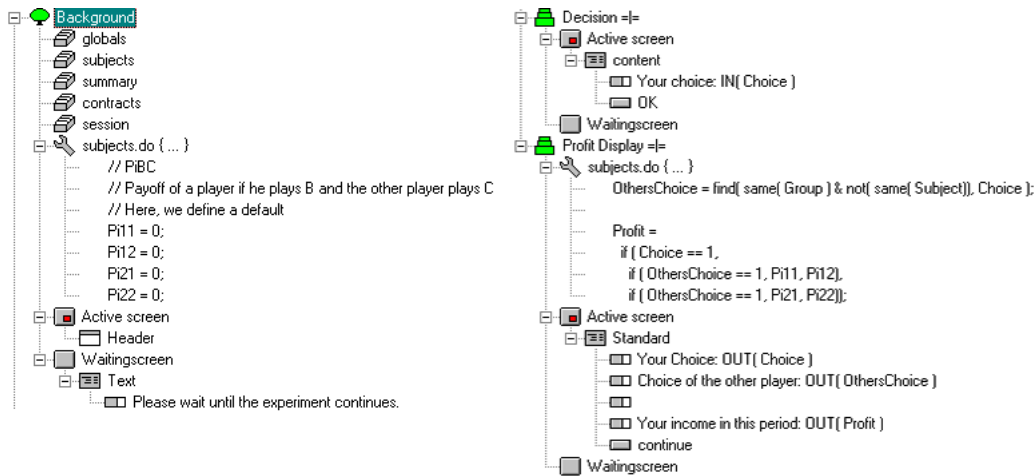
Figure 3.17. Stage tree of the battle of the sexes game

In this experiment, the players may have different roles. However, they differ only with respect to the parameters. They have different payoff functions. We implement this in the following way. First, we define in the Background defaults for the payoff matrix. The variable $Pi\_BC\_$ is the payoff of a player if he plays as $B$ and the other player plays as $C$. For instance, $Pi21$ is the payoff the player gets if he plays as 2 and the other player plays as 1. The actual parameters for the different subjects are then defined in the parameter table as shown in the following figures.
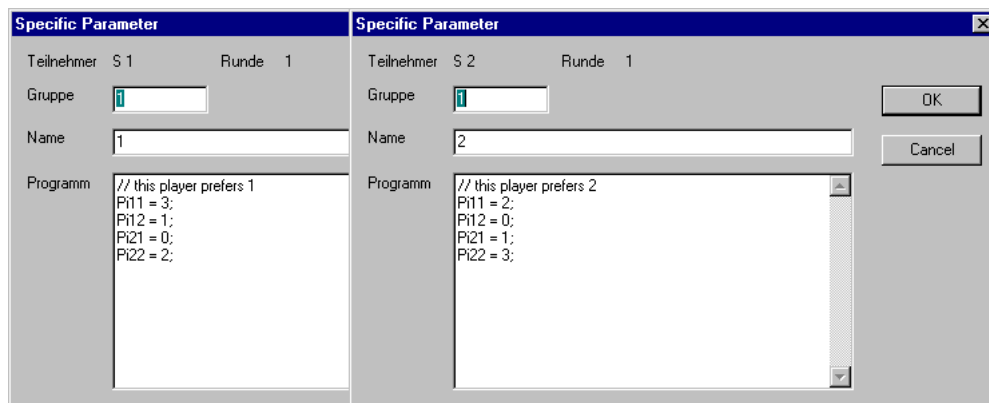


Figure 3.18. Parameter table of the battle of the sexes game

To calculate the payoff, we need to know the choice of the other player. Because the other player's variable $Choice$ is not in one's own record, we have to apply a table function. The function $find$ runs through the table and returns an entry as soon as some condition is met. The other player's record has the properties that the $Group$ variable has the same value as ours and the $Subject$ variable is different. This is expressed in the command:

```
OthersChoice = find( same( Group ) & not( same( Subject ) ), Choice );
```

Finally, the profit is calculated with a nested if function.

```
Profit =
    if ( Choice == 1,
        if ( OthersChoice == 1, Pi11, Pi12 ),
        if ( OthersChoice == 1, Pi21, Pi22 ) );
```

In this payoff function, the parameters in the parameter table are used.

There are many solutions to solve a particular problem. Let us also sketch an alternative solution: We could define a variable `Type` for the two types of players. Then we define the two payoff matrices in the globals table. In the parameter table, we only set the `Type` variable. Finally, the payoff calculation is a bit more complex because we have to distinguish between the two types.

### 3.3.8 Example: First price sealed bid auction

In a sealed bid auction, the bidders place secure bids, i.e. bids that cannot be viewed by the other bidders. At the close of the sealed bid auction, all bids are available for review and the highest bid is evaluated for award.

In this type of auction, the problem is to avoid ties, i.e. the possibility that two or more sealed bids are the same. To avoid ties, you can add a random number – smaller than the resolution of an entry – to each bid:

```
BidPlusRandom = Bid + random() * .8;
```

After having solved the problem of the ties, we need to determine the maximal bid to find out which subject is awarded:

```
MaximalBid = maximum ( BidPlusRandom );
```

The two variables `BidPlusRandom` and `MaximalBid` must be calculated in two separate programs of the subjects table since programs are carried out completely for an entire table, i.e. for each record of a table. In our example, the variable `BidPlusRandom` should be calculated before the variable `MaximalBid` is determined to get a correct result.

Then the winner is:

```
Winner = if ( MaximalBid == BidPlusRandom, 1, 0 );
```

If you wish to keep track of winners in a table, define a winnerhistory and a winner table in the Background. When the subjects have made their bids, the winner table needs to be updated. To do this, you write a new program into the subjects table (for example in the stage of the profit display):

```
winners.new {
    Subject = :Subject;
    NumWon = winnerhistory.count( Winner == :Subject );
}
```

Please see the reference manual for random variables. Another solution to sealed bid auctions and tie-breaking is explained in the next section.

### 3.3.9 Example: Private value second price auctions

**The experiment**

The participants bid for a good, which has a private value to them. If they get the good, their income will be their private value minus the price they pay. To get the good, they simultaneously submit an offer. The person with the highest bid is the winner and gets the good. The winner has to pay the second highest bid. (If the two highest bids are equal, he has to pay this highest bid). In this example, all members of a session will be in one group (though naturally, the program can be modified to have multiple groups).

**The solution**
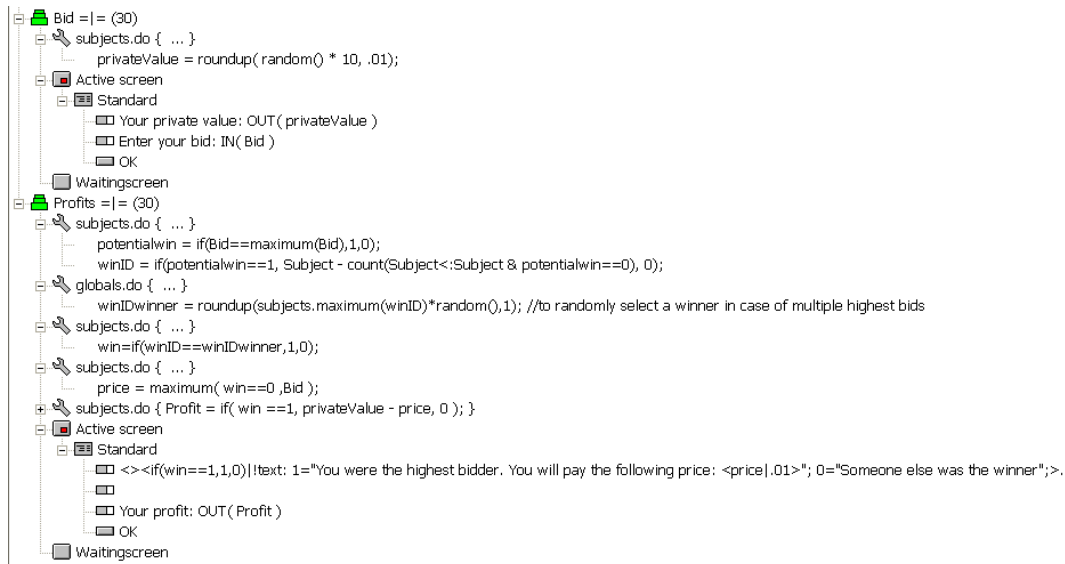


Figure 3.19. Stage tree of a private value second price auction

The program is shown in Figure 3.19, "Stage tree of a private value second price auction". In the beginning of the first stage, we generate the subjects' private values for the good. In the stage Bid subjects then enter their bid values. The profit calculation in the Profits stage has 5 steps. First, we calculate `poten-tialwin` to keep track of the subjects who made the highest bid. Since it is possible for several people to have the highest bid, we allow for the possibility to pick one random winner among all of the highest bidders. Next, we use `winID` to give each potential winner a unique ID. With `winIDwinner` in the globals table, we select a random number. In the third program we find the subject with the `winID` value equal to the random `winIDwinner` value, and give this subject the value `win` equal to one. Everyone else gets a `win` value of zero, so we have selected our winner. Last we must find the price that the winner will pay and calculate her profit. The `price` is the highest price among those who have not won, and the `Profit` for the winner, as stated above, is equal to the private value minus good's price. All other subjects make no profit. On the subjects' screens we show whether they were a winner or not and what

their profit is. Note: in this example there are variables inserted into text. How to do this will be explained in more detail later.

### 3.3.10 Programming group matching

Defining the matching with the menues has the disadvantage that the command has to be reapplied if the number of subjects or periods changes. This is for example relevant when you test the treatment with few subjects. Since the `Group` variable is a normal variable, you can also define it in a program. Below you find examples of partner and stranger matchings. If you define the groups in a program, you should set the number of groups in the backgroud to 1 in order to avoid an error message.

```
NumInGroup = X; //whatever you like
N = subjects.count();
NumGroups = N / NumInGroup; //whatever you like

// Partner matching 111222333444
subjects.do {
    Group = roundup(Subject / NumInGroup, 1);
}
// Partner matching 123412341234
subjects.do {
    Group = 1 + mod(Subject - 1, NumInGroup);
}
// Stranger matching
repeat {
    subjects.do {
        R = random();
    }
    subjects.do {
        Rank= count(R >= :R);
    }
} while(subjects.sum(Rank) - N * (N + 1) / 2 > .5); //repeat in case of ties
subjects.do {
    Group = 1+rounddown((Rank - .5) / NumInGroup, 1);
}
```

### 3.4 Screen layout

So far, you know how input is made with the help of input items in standard boxes and you know how to display variables. In this chapter an overview of more sophisticated layouts is given. In the *real* 'battle of the sexes' game, for instance, the players can choose between 'boxing' and 'opera'. In the implementation above, the subjects had to enter a number for their choice. In this section, we will present how input can be made with different forms of user interface elements such as text input, radio buttons, check boxes, and sliders. Furthermore, we show how information can be displayed and arranged on the screen.

### 3.4.1 Layout in the small: the item

Items are used for displaying and reading in variables. An item contains the name of the variable and information on how to display it. We call an item an *input item* if the checkbox Input is checked. In this case, subjects must make an entry. We call an item an *output item* if the checkbox Input is not checked. In this case, a variable will be displayed. Variables contain numbers or strings. Numbers can be displayed in different forms. These forms are defined in the Layout field of the item dialog. If a number, a variable, or an expression is entered here, then the value of that field determines how the variable is rounded when it is displayed. If, for instance, `0.2` is entered in Layout, and the variable contains the value 52.31, then 52.4 is displayed because 52.4 is the multiple of .2 nearest to 52.31. The value of the variable can also contain a code for displaying text, e.g., in the battle of the sexes game 1 could mean boxing and 2 could mean opera. We can display these words by using a *text layout*. In the Layout field, we enter the text:

```
!text: 1 = "boxing" ; 2 = "opera";
```

The exclamation mark indicates that the field does not contain simply a number (or an expression that represents a number). The word `text` is the form of output to be displayed. It instructs z-Leaf to display one of the words "boxing" or "opera" (without the quotation marks) depending on the value of the variable. If the value is 1, then boxing is displayed; if the value is 2, then opera is displayed. (To be precise, if the value is closer to 1 than to 2, then boxing is displayed; otherwise, opera is displayed.) There are other options than text. The radio option allows you to display radio buttons. So, the following layout

```
!radio: 1 = "boxing"; 2 = "opera";
```

displays two radio buttons labeled with boxing and with opera. Depending on the value of the variable, one of the two radio buttons is selected. These kinds of layout options can also be used for input items. For an input item with the text layout as above, the input has to be entered in textual form. If "boxing" is entered, the variable gets assigned a value of 1. If "opera" is entered, the variable gets assigned a value of 2. If anything else is entered, such as "I do not know" or "BoXing", then an error message appears that says that only certain values are accepted. With the radio option, two radio buttons labeled with boxing and with opera are displayed. Selecting one of them sets the variable to the corresponding value.

The following options are available: text, radio buttons, line of radio buttons, check box, slider, horizontal scrollbar, and push buttons as shown in the following table. All options except the push button option can be used for input items and for output items. The push button option can only be used for input items because a push button cannot be displayed differently as selected or unselected. How the different options are programmed is explained in the reference manual.

### 3.4.2 Layout in the large: Screen design and boxes

In experiments, subjects work for only one to two hours at the computer. They cannot gain much experience and should be able to understand screens very quickly. For that reason, the screen layout of z-Tree is rather static. The screens are built of fixed rectangular areas called *boxes* that can be placed freely on the screen. Besides the standard box with which you are already familiar, there are other boxes, for instance, a help box for explanations, a header box for information about the current period and remaining decision time, and a history box in which you can display information about earlier periods.

In the next sections, we first explain how boxes are placed on the screen. Then, we present some of the specialized boxes.

### 3.4.3 Placing boxes

Boxes are positioned one after the other on the screen. By defining size and/or distance to the margin as shown in Figure 3.20, "Placement of the box is defined in the box dialog", you can determine where the box appears. Any size can be given in screen pixels (p) or in percent.
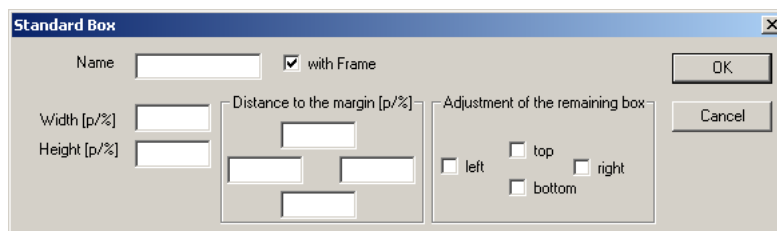


Figure 3.20. Placement of the box is defined in the box dialog

The positioning of boxes is always relative to the so-called *remaining box*. At first, the remaining box constitutes the whole screen. Later, the remaining box is adjusted according to the definitions of the boxes. For example, if you place a header box on the top of the screen and you want to place the rest of the boxes below this header box, then you can cut the top away. You just have to check the top checkbox of the Adjustment to the remaining box field in the box dialog. This means that the bottom of the header box becomes the top of the remaining box.

Width, height and distance to the margin are optional. Depending on which fields have been filled, the box is placed within the remaining box. Figure 3.21, "Box placement" shows all cases for the fields width (W), distance to left margin (L) and distance to right margin (R).
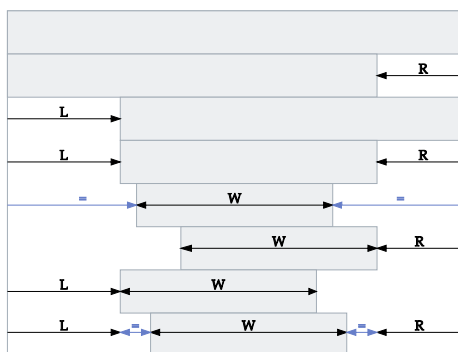


Figure 3.21. Box placement

The Name of the box is used for documentation only. In the with Frame field you define whether a line is drawn around the box. If there is a frame, then this frame is filled, i.e., if you draw a box with a frame over another box, then this box will be covered. Otherwise it is transparent.

In Figure 3.22, "Box placement example", we show definition and placement of four boxes. The first box is a header box. We define a distance from the top margin of zero and a height of 10 percent. The rest of the boxes should not intersect this box. So we adjust the remaining box and cut the top. The

second box is positioned at the left side of the remaining box with distances of 20 pixels to the margin. We do not modify the remaining box. Hence, the third box where we define width and height is placed centrally with respect to the lower part of the screen. In this box we adjust the remaining screen in a way that the bottom part is cut. The remaining box is then the rectangular area between box 1 and box 3. In the fourth box we specify no positioning. Therefore it fills this remaining box. It in particular intersects box 2. The dashed line is shown here only for illustration. In the actual screen, the part of box 2 that is covered by box 4 is invisible.
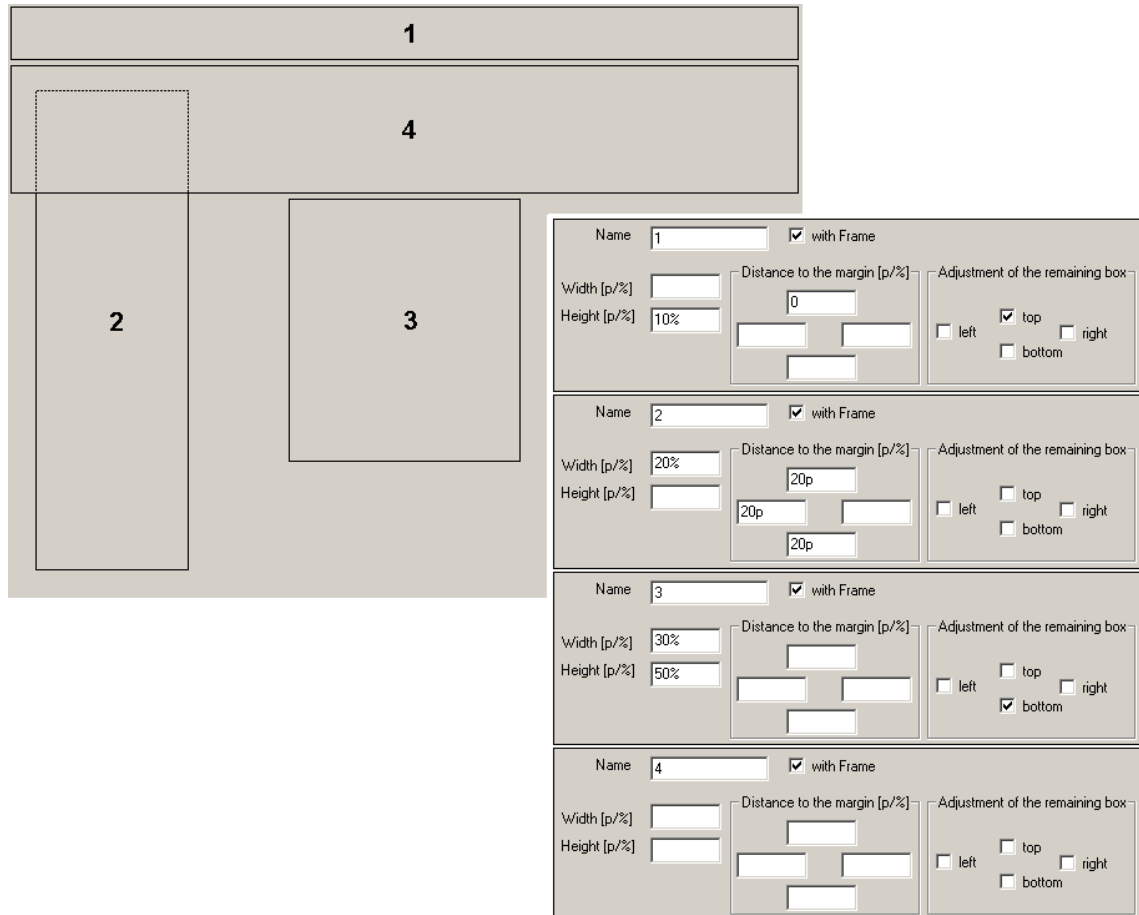


Figure 3.22. Box placement example

### 3.4.4 Basic box types

In this section, we present the basic box types. In these boxes you display texts and data from the subjects table.

**Standard box**

In a standard box, variables of the subjects table may be displayed or entered. The items are displayed from top to bottom. The window is divided into a label column (left) and a variable column (right). The variables are always displayed in the variable column. The label always appears in the left column if the variable is defined or if the variable consists only of an underscore ("_"). If the variable is empty,

the label is regarded as a title and is written centered over the whole window. If the label consists of more than one line, then it is aligned to the left. If an item is completely empty, it generates a blank line.
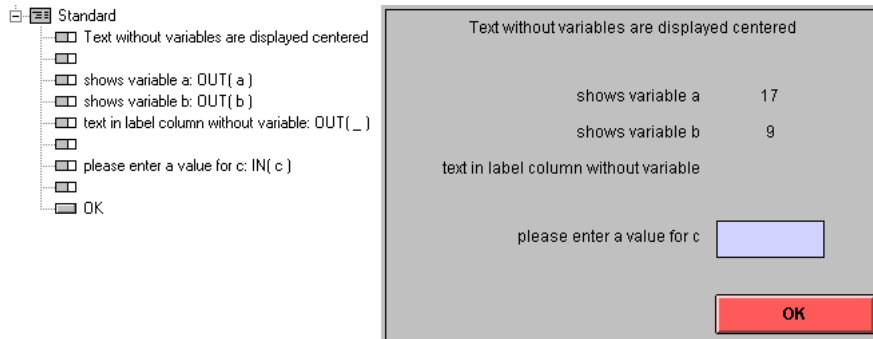


Figure 3.23. Example of the definition and the resulting layout of a standard box

## Grid box

Like the standard box, the grid box can contain items. Unlike to the standard box, items in a grid box are displayed in a table. Each item belongs to one cell in the table. If the item contains a variable, this variable is displayed, and labels are only used for error messages that convey to the subject in which field he or she has made a mistake. If an item does not contain a variable, then the label of the item is displayed. In the grid box dialog, you define the number of rows and columns and the order in which the items are filled in. You can fill the items column by column or row by row.
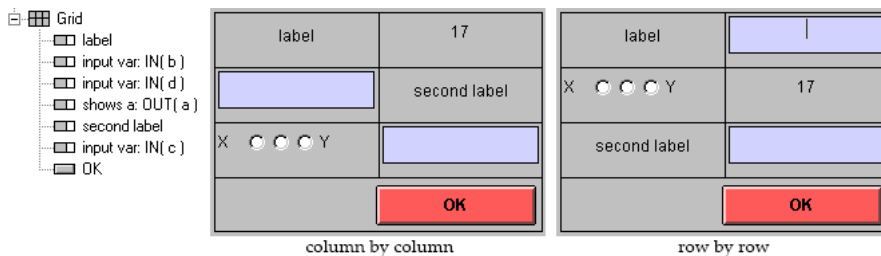


Figure 3.24. Grid box examples with column by column or row by row layout

## Header box

In the header box, period number and time are displayed as shown in the following figure. All information is optional and can be defined in the header box dialog.
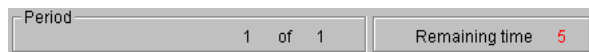


Figure 3.25. Header box example

## Help box

The help box displays text within a box. The size of the text is not restricted to the size of the box. If the help text is too long to appear in the area reserved for the help box, then a scrollbar appears. The help box can be labeled.
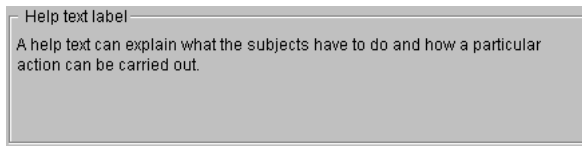
Figure 3.26. Help box example

**History box**

In a history box, the results from previous periods are listed in a table. A label row contains the labels. If the table is too long, a scrollbar appears. The current period appears at the end of the table. The scrollbar is adjusted in such a way as to make this line visible at the beginning.



Figure 3.27. History box example

**Container box**

If you place many boxes onto the screen, you may lose track of things. To avoid confusion, you can use container boxes to structure the screen elements. A container box has, besides the frame, no visual representation on the subjects' screens. It only defines areas of the screen within which you can place boxes.

Container boxes also make it possible to define determinants of your screen layout at fewer places. So, if you change the layout, you have to change it at relatively few places. In Figure 3.28, "Container box example", we would like to define the width of box 1 and 2 at only one place. With container boxes, this is easy. We define two container boxes, 12 and 34. We define the width in the container box 12 and cut off the remaining box at the left. Now container box 34 fills the region on the right. In boxes 1 and 3 we define the heights and cut the remaining box off at the top. In this way we have not entered a size specification at more than one place, and even if we should change something, the structural appearance of the screen remains the same.
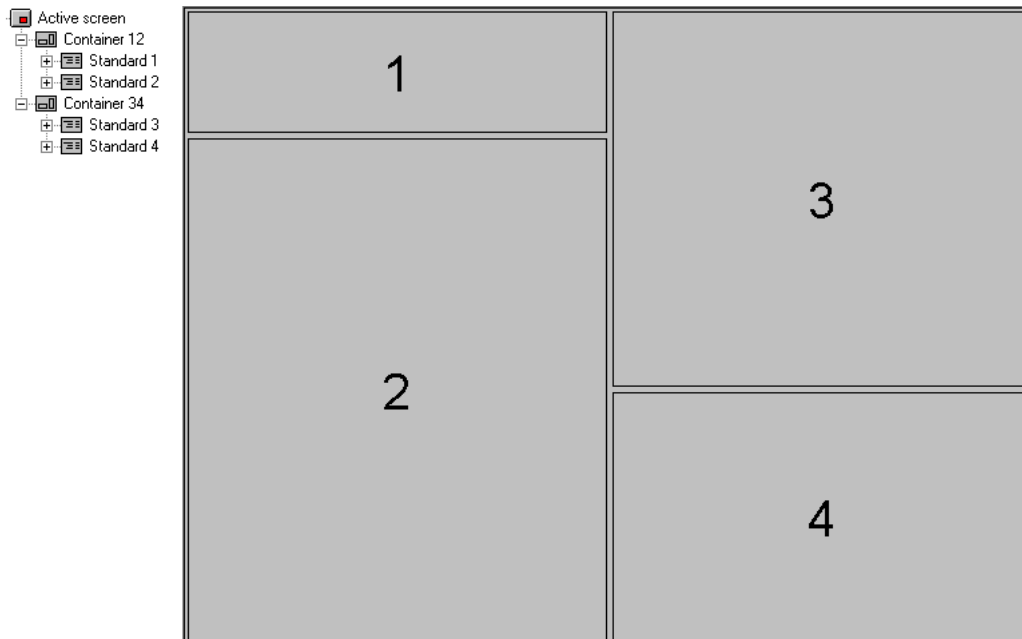
Figure 3.28. Container box example

If we move a box from one place to another inside the stage tree, it is moved *after* the box where the mouse button is released. By moving a box over the *icon* of a container box it is moved *into* the container box and positioned at its beginning.

**Calculator button box**

Defines a button as shown below which calls the Windows Calculator. It serves as a substitute for subjects who have forgotten their calculators.



Figure 3.29. Calculator button box example

**3.4.5 Button placement**

To confirm input and to conclude a stage, you use buttons. They can be placed into standard boxes and into grid boxes. By default they are placed at the bottom right corner. If you prefer another place in the box, then you can choose another option for the buttons position in the box dialog. If you have more than one button, then they are placed in a row at the bottom. The necessary amount of rows are created. In the box dialog, there are two fields where you can modify the button positioning. You can define where the first button is placed (Position) and where the subsequent buttons are positioned (Arrangement). In Figure 3.30, "Button placement example", we show how five buttons are placed with three different option combinations.
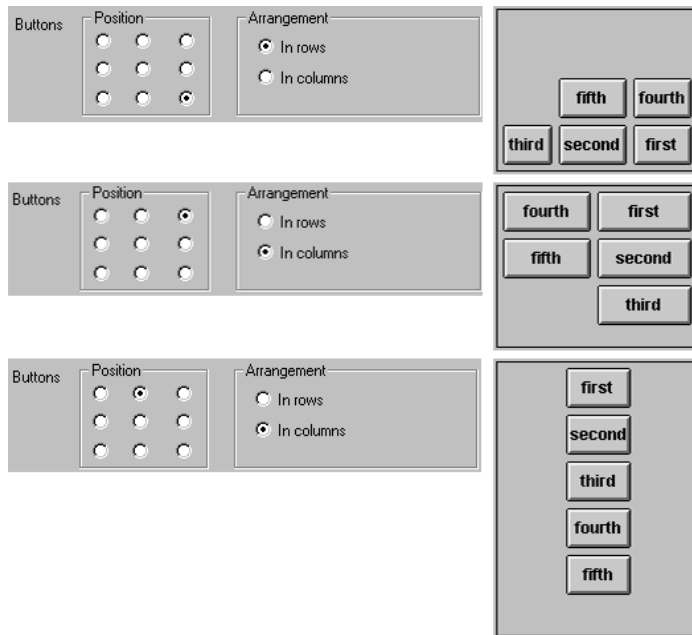
Figure 3.30. Button placement example

If you want to leave space in the size of a button without displaying a button, then use an underscore character ("_") as the name of the button.

### 3.4.6 Background layout

On every screen, the boxes in the Background are automatically placed first. For each screen, you can determine whether you want to use these background screens. You find the corresponding option (Background screen is being used) in the screen dialog. If this field is checked, then the windows of the background are placed first.

### 3.4.7 Insertion of variables into the text

Variables can also be inserted in the label and in the layout field of items, as well as in help and message boxes. Thanks to this you can, for instance, display a formula with the values inserted. Assume that we want to write "income = 23.5 points" where 23.5 is the value of the variable Profit. So that the variable does not stand alone on the right hand side, it needs to be integrated into the text. The example above is entered in the Label field in the following manner:

```
<>income = < Profit | 0.1 > points
```

The string "`<>`" at the beginning of the text indicates that there might be variables in the text. The variable and its layout appear in smaller/greater brackets, separated by a vertical line. In our example, 0.1 is the layout. The value of profit is therefore given to one decimal place.

The option !text: is also allowed as a layout. This works in exactly the same way as with output items. Example:

```
<>Your income is < Profit | !text: 0="small"; 80 = "large";>.
```

Depending on whether the profit is nearer to 0 or to 80, either "Your income is small" or "Your income is large" is displayed.

Variables can also be inserted into the strings of the "text" option. There you don't need to begin the string with a second "<>". If you want to define a layout in which, for negative values, text is written but for positive values, the number (to two decimal places) is written, you write

```
<>!text: -1 = "negative"; 1 = "<Profit | 0.01>";
```

This form has the disadvantage that you also need to write the name of the variable into the `Layout` field. However, you can omit the name of the variable. In this case the variable of the *item* is automatically shown:

```
<>!text: -1 = "negative"; 1 = "< | 0.01>";
```

This also works with variables inserted in the text. In the following text, the value of `Profit` is inserted at the place of `< | 0.01>`:

```
<>Your profit is < Profit | !text: -1 = "negative"; 1 = "< | 0.01>">
```

If you wish to insert the symbol < in the text, the symbol should be duplicated so that it is not confused with a variable expression.

Be careful when you use this option for items that can change their value. The width of an item is calculated when the screen is first displayed and it is not modified afterwards. So, if the value of a variable changes from 1 to 20, only 2 might be displayed. To avoid this problem, you can choose a sufficiently wide first value or place the item into a box with other items that are wider.

Note further that labels are evaluated only once, at the beginning of the stage. Variables in labels are not updated.

### 3.4.8 Text-formatting with RTF

In labels of items, in help boxes, and in message boxes, texts formatted with RTF can also be entered. The RTF format begins with "{\rtf " (with a blank space at the end) and ends with "}". In between is the text to which formatting instructions can be added. Formatting instructions begin with "\" and end with a blank space. If a formatting option is supposed to apply only to a certain range, then you can place this range in curly brackets.

For more complex operations it is best to format the text in a word processor and then export it as RTF. However, if you make the RTF code by hand, it will be shorter and much easier to read.

**Examples**

```
{\rtf \fs18 normal font size, \b bold, \b0 no longer bold}
```

normal font size, **bold**, no longer bold

```
{\rtf \fs18 Text {\i italic} no longer italic \par new line}
```

Text *italic* no longer italic
new line

```
{\rtf {\colortbl;\red0\green0\blue0;\red128\green128\blue0;}
\fs18 One word in \cf2 olive\cf1 , the rest in black.}
```

One word in olive, the rest in black.

**Combining RTF and inserted variables**

The insertion of variables is carried out *before* the interpretation of RTF. This makes conditional formatting possible as in the following example. When the variable BOLD is 1, "hallo" should be shown in boldface but otherwise in plain text.

```
<>{\rtf <BOLD|!text:0="";1="\b ";>hallo}
```

## 3.5 Sequential games

In a sequential game, not every subject has the same decision structure. In an ultimatum game for instance, only the proposers decide what to offer and only the responders accept or reject. Furthermore, you may want to allow simultaneous entry of these different decisions. If you apply the strategy method, then proposers and responders should be able to make their entries simultaneously. However, they make their entries in different screens.

In z-Tree, every treatment is defined as a linear sequence of stages. However, it is not necessary for all subjects to go through all stages and it is possible that not all subjects are always in the same stage. How this is achieved is explained in this section.

### 3.5.1 Sequential moves

In z-Tree, the variable Participate in the subjects table determines whether a subject enters the screens of a stage or not. If this variable has a value of 1 then the subject enters that screen, i.e., the corresponding active screen appears on the subject's screen. Before the programs in a stage are executed, this variable is set to 1. So, if you do nothing, then all subjects enter that stage. If however this variable is zero, then the subject does not enter the stage. The subject automatically proceeds to the waiting state of that stage without seeing the screens, i.e. the waitingscreen being shown is not replaced by the waitingscreen of the stage.[3] How can you change the value of Participate? This variable is a normal variable of the subjects table and you can set it to zero or one depending on a condition.

In the ultimatum game example, the proposers move first, making an offer decision. After this, the responders decide whether or not they want to accept the offer or not. Therefore, we define a Proposer Decision stage and a Responder Decision stage. Only the proposers go through the former, and only the responders go through the latter. To this end, we write the following line in a program of the subjects table in the Proposer Decision stage:

---

[3]This is necessary because you may want to omit more than one stage. In this case the change from one waiting stage to the next should be invisible to the subjects.

```
Participate = if( Type == PROPOSERTYPE, 1, 0 );
```

This line sets the variable `Participate`. If it is zero, the subject does not go through this stage. He or she directly reaches the waiting state of that stage, without the display changing. To the subject, it looks as if he or she were still in a previous stage. For the Responder Decision stage, we use the program

```
Participate = if( Type == RESPONDERTYPE, 1, 0 );
```

It is important to know that all programs at the beginning of a stage are executed for *all* subjects. Only when all programs are finished is the variable `Participate` checked. The variable `Participate` does not determine program execution. It determines only what screens are shown.

> Always write the participate statement in the following form:
>
> ```
> if( ConditionForEntering, 1, 0 )
> ```
>
> If you do not follow this convention, then you always have to pay attention to the second and third arguments of the `if` statement which can easily be overlooked.

### 3.5.2 Simultaneously in different stages

In this section, we show how to program a simultaneous stage. Consider the example of an ultimatum game with the strategy method or any game in which different types receive different feedback – for instance in the profit display.

In each period, the subjects go through all stages, one stage after the other. In each stage, they first arrive at the active state of the stage. In this state, the subjects see the active screen of this stage. In the clients' window, the active state of a stage is designated as "`". The active screen is left by means of an OK button or a time-out. When this happens, the waitingscreen appears and the subject arrives at the waiting state of that stage which is designated as "`". Whether or not the subjects can begin with the next stage depends on how the options are set in the dialog of the next stage. The first of these options is the Start option. It determines the precondition for the subjects to enter the stage. The first two options are the options most often used. Wait for all is the default. If it is the value of the start option, then the subjects cannot enter the stage until all subjects have finished the previous stage. If the option is set to Start if possible, then there is no restriction at all. So, as soon as a subject has finished the previous stage, she can enter the stage. The option Start if… allows you to enter a condition. This feature is only needed for complex move structures. Most experiments can be programmed with the first two options.

The start option allows you to define stages that are executed simultaneously. Consider a two stage sequential game where the strategy method is applied, i.e., the second mover has to make a decision for every possible choice of the first mover. In this case, the second and the first movers can make their decisions simultaneously. We define this in z-Tree as follows: We define two stages, one for the first mover, and one for the second mover. We set the `Participate` variable in the two stages so that only the first movers enter the first mover stage and only the second movers enter the second mover stage. Finally, we select the individual start option in the second stage. So, the second movers will not participate in the first stage and because they do not have to wait to enter the second stage, they enter the second stage (essentially) at the same time as the first movers enter the first stage. Figure 3.31, "Example

of a two stage sequential game" shows this situation. At the end of the stage name, there is a symbol that shows the start option.

```
⊟ 🔋 first mover =|=
   ⊟ ⚒ subjects.do { ... }
           Participate = if( Type == FIRSTMOVERTYPE , 1 , 0);
   ⊞ ▣ Active screen
        ▢ Waitingscreen
⊟ 🔋 second mover -=
   ⊟ ⚒ subjects.do { ... }
           Participate = if( Type == SECONDMOVERTYPE , 1 , 0);
   ⊞ ▣ Active screen
        ▢ Waitingscreen
```
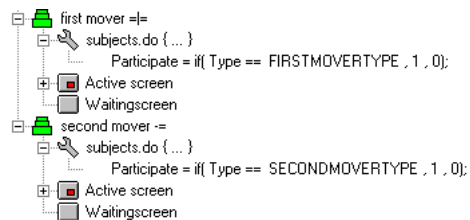
Figure 3.31. Example of a two stage sequential game

### 3.5.3 Ending a stage

For each stage, the time available to the subjects is fixed in the Timeout field. The expression entered is calculated in the globals table. If different stages should have the same time-out times, define a variable in the globals table and enter the name of this variable in the time-out field.

After the time set for a stage has run out, the continuation of the game depends on the option set in the stage. The default behavior depends on whether entries have to be made in this stage or not. If the subject does not have to make any entries on this screen, then the stage ends when the time set for the stage has run out, i.e., the subject arrives at the waiting state of the stage and may go on from there. If entries have to be made, the time displayed is simply a guideline. When the time has run out, a message saying "Please make your decision now" appears. However, the game does not continue until the entries are made. If no timeout should occur, you can enter a negative number. The subjects will then have an unlimited amount of time at their disposal. If the time-out is set to zero, then the subjects will arrive at the waitingscreen straight away.

If you do not want to leave a stage automatically, then you set the option to No in the option Leave stage after timeout. If you want to leave the stage even if no input has been made, then you enter Yes in this option.

⚠ If you use the Yes option in Leave stage after timeout, you want to set the value of input variables to the appropriate default value.

### 3.5.4 Example: Ultimatum Game

Player 1 (the proposer) can propose a division of a pie of 100. Let the proposal be (share1, share2). Player 2 (the responder) can accept or reject this division. If player 2 should accept this proposal, then both players receive their shares. If player 2 rejects, then both receive nothing.

As shown in the following figure, we define four stages for this treatment: The decision of the proposers, the decision of the responders, the profit display of the proposers and the profit display of the responders. We now go through the main steps in programming this experiment.

Figure 3.32. Example of an ultimatum game

First, we define the constants PROPOSERTYPE and RESPONDERTYPE. We use these constants as names for the numbers 1 and 2. Then, we define the variable Type. We set it here to an illegal value and define the correct value in the parameter table. To make the treatment more flexible, we define the size of the pie as a variable. We call this variable M. The variable IncOffer contains the increments that are allowed when making offers. We allow any integer number and therefore we set this number to 1. In the Proposer offer stage, we set the Participate variable so that only proposers make offers.

In the Responder acceptance stage, we have to copy the proposer's offer into the responder's row of the subjects table. We do this by defining a new variable Share. It contains both subjects' shares as suggested by the proposer. As you can see in the formula, it is M - Offer for the proposer and it is the offer of the proposer for the responder. To get the offer of the proposer, we use the following find expression

```
find( same( Group ) & Type == PROPOSERTYPE, Offer )
```

In the Responder acceptance stage, the responder can either accept or reject. We use radio buttons for input, i.e., in the Layout field of the Accept item, we write:

```
!radio: 1="accept"; 0="reject";
```

In the Proposer profit stage we calculate the profits for both types. The interesting command is in the program where we copy the Accept variable from the responder row to the proposer row:

```
Accept = find( same( Group ) & Type == RESPONDERTYPE, Accept );
```

With this line, we overwrite the value in the `Accept` variable for both types. Be cautious with overwriting, you may destroy valuable data. Here, we do not destroy anything. For the responders, we overwrite the value with the old value, i.e., we do not change it. For the proposers, the value of `Accept` had no meaning before. The use of this kind of overwriting makes a program easier to read because it makes a variable that was available only for certain types available to all types in the group.

We place the calculation of the `Participate` variable into a separate program. We have to use this program in two places and if we separate it, we can simply copy and paste the whole program.

The profit display is placed into different stages for proposers and responders, as they each have a different display. In the two stages we set the `Participate` variable in such a way that only the proposers enter the proposers' profit display and only the responders enter the responders' profit display. Finally, we change the start option in the responders' profit display stage to Start if possible. If we omit doing this, the responders only receive the profit display after the proposers have been shown their profit. This creates an unnecessary delay.

## 3.6 Continuous Auction Markets

### 3.6.1 Concepts

In the treatments discussed so far, the basic structure was very simple: In each stage the players enter their decision, confirm the decision with a button and then wait until the experiment can proceed, i.e., they wait until they can proceed to a next stage. With this mechanism it is possible to implement any kind of normal form or extensive form game. However, many economic experiments use market institutions that cannot be implemented as games. Consider for instance a double auction. In this market institution, sellers and buyers can make offers. These offers are shown to all market participants. The buyers can accept the sellers' offers and the sellers can accept the buyers' offers. This institution differs from the treatments we have programmed so far in the following ways:

– The subjects do not make a *predetermined* number of entries. We do not know in advance how many offers will be made.

– We do not know in advance in what order offers are made, i.e., we do not know who will be the next to decide.

– Even subjects who do not make an entry are informed about other players' decisions, i.e., all offers are shown to all subjects.

– The auction is automatically terminated after a certain time.

To deal with continuous auctions, we use the following concepts in z-Tree:

– In the stage tree dialog, there is an *option for terminating a stage automatically when the time has expired*. This also changes the default behavior of buttons. Buttons no longer conclude the stage. So, subjects can make an indefinite number of entries.

– We need a table without predefined number of records. An example is the contracts table. It contains an indefinite number of records. In the contracts table we store the subjects' offers.

– There are new box types, the *contract creation box*, the *contract list box* and the *contract grid box*. These boxes allow subjects to make, select, view, and edit offers. By using several boxes, it is possible to do different things on one screen – in one stage. For example, subjects can make offers in one box and accept offers in another box.

– By *placing programs into buttons*, it is possible for subjects to take different *actions* based on the same data. For example, the entry of a price can mean sell for this price or buy for this price.

In the following sections, we go through the development of a simple double auction. In this double auction, sellers can sell a product that has a cost of zero to them and a value of 100 to the buyers. Each subject can trade at most one item. We first show how the entries of the subjects are stored in the contracts table. Then, we show how to make the box for making an offer, the box for displaying the offers, and the box for selecting an offer.

### 3.6.2 A double auction in the contracts table

In an auction, the records of the contracts table contain the offers made by sellers and buyers. If an offer is accepted the corresponding record is updated. This information is stored in the variables `Seller`, `Buyer`, `Creator` and `Price`: If a seller makes an offer, the variable `Seller` is set to the `Subject` variable of this subject.[4] Since this number is unique, we also say that this is the subject ID. The variable `Buyer` is set to –1 which means that this is an open offer. The variable `Price` is the price the seller enters as her offer. When a buyer accepts a seller's offer, the number –1 in the variable `Buyer` is replaced by the subject ID of the buyer. The analogous changes to the contracts table are made if a buyer makes an offer and if a seller accepts a buyer's offer. As you have seen, it is easy to map the creation and acceptance of offers in the contracts table.

If an offer is accepted, it is not sufficient to mark the offer as accepted. Think of a seller who makes decreasing offers whose lowest offer is eventually accepted. It is clear that the other open offers of this seller are no longer valid. For instance, it is possible that this seller has no more goods to sell. Therefore, we have to delete the outdated offers. Obviously, we do not want to actually delete these offers in the table, since we want to have a record of all actions taken by subjects. We only want to mark them as deleted. We do this by replacing the –1 in the variable `Buyer` by –2. Using the number -2 is somewhat arbitrary, and codes an entry as deleted very simply because we have decided that -2 means "deleted". In this way, we keep all the information about what happened during the auction.

In a double auction, there remains one more problem. If an offer is accepted, we no longer know who made the offer, because in such an offer both the `Buyer` and the `Seller` variables are positive. We solve this problem by using a variable `Creator` that contains the subject ID of the subject who made the offer. We could set this variable when the offer is accepted. However, it is more natural to set it when the offer is made. Finally, one should also be able to reconstruct the temporal structure, in particular the sequence of offers. The easiest way to achieve this is to update a counter of the number of interactions (offers and acceptances) in the globals table or to record the time using the function `gettime()`.

The following table summarizes the different situations that occur in a double auction. The value of the variables allows us to reconstruct what happened during the auction.

---

[4]This is not the only way to program an auction. However, this is very convenient. It is in particular a good practice to always use the same set of variables. It makes it much easier to understand your treatments.

| | Seller | Buyer | Creator | Price |
|---|---|---|---|---|
| Open offer of seller | ID of seller | -1 | ID of seller | Offer |
| Accepted offer of seller | ID of seller | ID of buyer | ID of seller | Offer=Trading price |
| Offer of seller that is no longer valid | ID of seller | -2 | ID of seller | Offer |
| Open offer of buyer | -1 | ID of buyer | ID of buyer | Offer |
| Accepted offer of buyer | ID of seller | ID of buyer | ID of buyer | Offer=Trading price |
| Offer of buyer that is no longer valid | -2 | ID of buyer | ID of buyer | Offer |

### 3.6.3 Preparing the layout for the auction

Before we start to define the boxes, let us discuss the layout. Because the layout of sellers and buyers are very similar, we only discuss the layout for the buyer. The buyer can make offers, he can view the offers of the other buyers and in a third box, he can view the offers of the sellers and accept them. We will arrange the screen as shown in Figure 3.33, "Sketch of box arrangement for an auction".
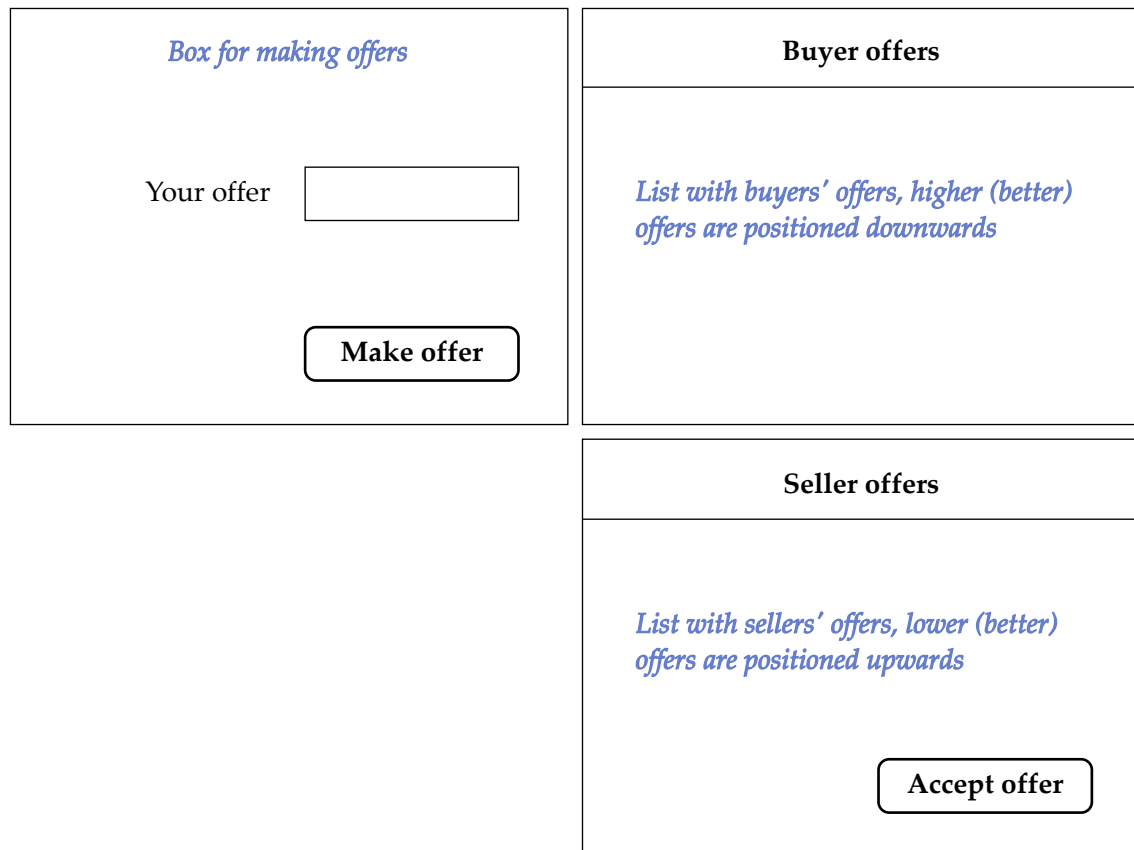


Figure 3.33. Sketch of box arrangement for an auction

The box for making the buyers' offers and the box of the buyers' offers are positioned next to each other, side by side.

### 3.6.4 Making an offer

In this section, we describe how to design a box for making an offer. Offers are made in a contract creation box. You insert a new contract creation box as you do any other box from the menu. The dialog of the contract creation box contains some fields that are specific to this type of box. These options are explained in detail in the reference manual. For this example, the options are set properly by default. The only thing that you have to change is the positioning.

Similar to the standard box, contract creation boxes can be filled with (input) items and buttons. In our contract creation box, the offer must be entered, i.e., we place an input item for the variable `Price` into the box. To confirm the offer, the subjects have to press a button. This button is also placed into the contract creation box. When a subject presses this button, a new record in the contracts table is created. The data in the input items, i.e., the data in the variable `Price` is stored in this record. How do we fill the other information into the record? We do not want the subject to have to make an entry for all the information in this new record. This seller should not have to enter the `Seller` variable, for instance. In general, this subject does not even know her subject ID. We want data that has no direct meaning to the subjects to be entered automatically. The solution to the problem consists of a program that is placed into the button. Such a program is executed when the button is pressed. It is easy to write the program that sets the Buyer variable to -1:

```
Buyer = -1;
```

But how do we set the `Seller` variable? How can we access the ID of the subject who pressed the button? The solution uses the scope operator. The program in the button is executed in a *scope environment*. When a program runs in the subjects table, we can access the globals table with the scope operator. Similarly, if we run a program in the button of a contract creation box, we can access the record of the subject who pressed the button with the scope operator. (A double scope operator reaches the globals table.) Now it is easy to set the variables `Seller` and `Creator`:

```
Seller = :Subject;
Creator = :Subject;
```

Note that we have to define the table in which the program should run. We want to run it in the newly created record of the contracts table. So, in the program dialog, we select the contracts table.[5]

### 3.6.5 Viewing offers

The open offers are shown in a contract list box. Such a box shows a part of the contracts table. The items that are contained in the box define which columns are shown. A condition defines which records are displayed. Since we want to show all open offers, the condition is

```
Buyer == -1
```

Because this is an expression, i.e., something that has a value, it does not end with a semicolon. The offers should be sorted according to prices. Because the seller should decrease the offers over time, we display decreasing prices. Hence the sorting expression is

[5]Different to a program at the beginning of a stage, a program in a button is not necessarily executed for all records of that database.

```
-Price
```

The minus sign makes the decreasing order. For increasing order, we do not have to use the plus sign, because increasing order is the default. By the way, you can also interpret the order created by `-Price` as increasing order with `-Price`.

### 3.6.6 Do-statement

As introduced in Section 3.2.5, "The do statement", for an auction we require a "do-statement". With the do-statement calculations can be carried out in all records of a table. With `do { statements }` the program `statements` is executed for all records in the current table. As in the table function, it is possible to precede the do-statement with a table name. By doing this, calculations are carried out in the table in question. As in the table functions, you may also use the scope operator here.

The do-statement is of particular importance in an auction environment because we are navigating between multiple tables. For all of the examples until now, we worked primarily in the subjects table. In an auction, however, there is the subjects table, for information specific to people, and the contracts table, where the market transaction information is stored. When a subject presses a button for instance, this action may be automatically associated with either the contracts table or the subjects table. Most likely though we will need to modify some information in both tables, leading to the use of the do-statement as seen in the next section.

### 3.6.7 Accepting an offer

To make it possible to accept an open offer, we only have to add a button to a contract list box. As in the contract creation box, we have to put a program into the button. This program marks the offer as bought (and sold). As in the contract creation box, we can access the own record with the scope operator. Therefore, we need the statement

```
Buyer = :Subject;
```

With this line the offer automatically disappears from the list of open offers since now, `Buyer` is different from -1. However, we are not yet through. First, we have to delete the offers that are no longer valid, i.e., the open offers of the buyer and seller of this offer. After the statement above, we put

```
contracts.do {
    if ( Buyer == :Buyer & Seller == -1 ) {
        Seller = -2;
    }
    if ( Seller == :Seller & Buyer == -1 ) {
        Buyer = -2;
    }
}
```

First, with `contracts.do` we proceed through the contracts table. As in a table, the scope operator allows us to access the own record. In this case, it is the record we have selected. So, the condition `Buyer == :Buyer & Seller == -1` is satisfied for any offer of the buyer who pressed the button that is not

yet accepted. We mark these offers as deleted by setting `Seller` to –2. These offers then disappear from the list of open offers.

In general, we have to put more statements into the program of an accept button: We must update the stock of goods and money of the players to reflect the purchase. We could for instance assume that the variables `Goods` and `Money` contain the players' quantities of goods and money. In this case, we add the following statements to the program:

```
subjects.do {
    if ( Subject == :Buyer ) {
        Goods = Goods + 1;
        Money = Money - :Price;
    }
    if ( Subject == :Seller ) {
        Goods = Goods - 1;
        Money = Money + :Price;
    }
}
```

This program ensures that the buyer receives one good and the seller receives the money.

### 3.6.8 Checking subjects' entries

Often, there are restrictions on the offers that can be made or accepted. Incurring debts is often not allowed. Furthermore, one often requires offers to improve over time, i.e., buyers must make a higher bid than the most recent highest bid. Conditions such as this can be implemented by putting a stage tree element called a *checker* into the buttons. When a button is pressed, the condition in the checker is checked. If the condition is not met, the input can be rejected.

Consider first the short-selling restriction, meaning that the seller cannot sell goods he does not own. The condition that expresses this is simply `:Goods>=1`. This means that the seller has at least one good to sell. This expression is entered into the Condition field of the checker dialog as shown in Figure 3.34, "Checker dialog".
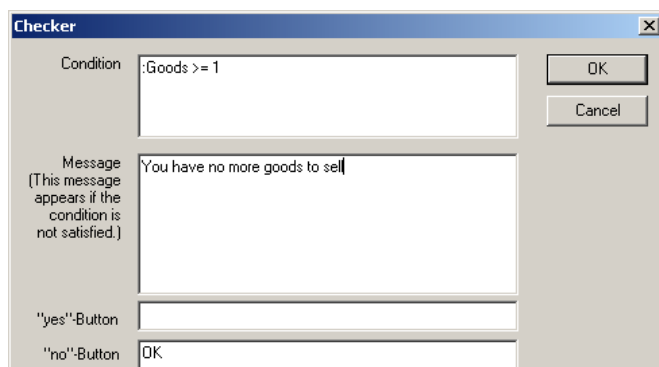


Figure 3.34. Checker dialog

If the check fails, a message appears on the subject's screen. The text of this message is "You have no more goods to sell" and is entered into the Message field. The last two fields are the names of buttons

of the dialog. If the subject presses the "yes"-button, the input is accepted and the text appears only as a warning. If the subject presses the "no"-button, the input is not accepted. In this case the condition is mandatory. Since there is only a "no" button, the subject cannot disregard the condition in the message. When the checker is created, it can be placed into the contract creation box as well as into the contract list box where an offer is accepted.

Then we create another checker for the seller side. It is the improvement rule that is implemented as

```
Price > contracts.maximum( Seller == -1, Price )
```

Note that it is necessary to prefix the "maximum" function with the name of the contracts table. In the contract creation box, a program as well as a condition is executed only in the new record, not in the contracts table as a whole.

> In versions earlier than 3.2.4, it was necessary to ensure that a selected offer is still available. This is now automatically checked.

**Where do the messages appear?**

In general messages appear in a new window and the subject has to click OK in order to close the window. This has the advantage that it catches the subject's attention. As an alternative, you can insert a message box in which the messages are displayed. In this case, the subject does not have to close the messages window but you risk that some subjects overlook such messages.

**Auctions within groups**

If you want to conduct auctions within groups, you have to set a group variable in the contracts table when you create contracts, and you have to restrict the display of the recods to those in the own group.

**3.6.9 Auction stages in detail**

We call a stage with an automatic timeout an auction stage. An auction stage forms an auction with all the following auction stages whose start option is "start if possible". Such an auction is started and ended simultaneously for all subjects. Before the stages start, all programs of these stages are executed. This is necessary in order to determine which subjects will participate in which stages.

Example: Though buyers and sellers in a double auction take part in the same auction, they are shown different screens. To this end, we define two stages: The first, an auction stage for the sellers, and the second, an auction continuation stage for the buyers. These two stages are started simultaneously, the first for the sellers, the second for the buyers. For that we insert the following program lines into the two stages:

```
Participate = if( IsSeller==1, 1, 0 ); // into first auction stage
```

```
Participate = IsBuyer; // into second auction stage; short variant
```

Generally, auction stages are concluded when the time has run out. There are two exceptions, however:

– If the variable AuctionStop is set to 1 in the globals table, the auction is terminated immediately. This is useful, for example, when all possible transactions are exhausted.

– If the variable AuctionNoStop is set to 1 in the globals table, the auction is not concluded even if the time is up.

In our example, we have to set the options for the sellers and the buyers in the stage dialog. For the sellers, we set Wait for all in the Start group and Yes in the Leave stage after timeout group. For the buyers, we choose Start if possible and Yes.

### 3.6.10 Creation of new records in the program

In the contracts table, new records can also be added with the "new" statement. The syntax is

```
table.new{ statements }
```

New records can only be entered into the contracts table and into user defined tables. The statements in curly brackets can contain initializations of the record. The following example is taken from the program of the subjects table of a certain stage. When a user reaches this stage, a record is added in the contracts table. In this record, the variables `Seller`, `Buyer` and `Price` are fixed. The value `Buyer` is the subject ID of the subject for which the program is run.

```
contracts.new {
    Seller = -1;
    Buyer = :Subject;
    Price = 25;
}
```

### 3.6.11 Adding good properties

Assume the subjects have to specify some property, such as quality, of the good traded after the double auction has ended. If each subject trades only a known number of goods, you can copy the information into the subjects table. The subject can then enter this information in the subjects table – for instance in an ordinary standard box. There is however an easier way of doing this. In a contract grid box, you can display a part of a contracts table and unlike to the contract list box, input variables can be added for *each* record.

So, you can create a contract grid box that displays the records of the contracts table that correspond to the trades of a subject. By adding an input item with a variable `Quality`, the subject can enter a quality to each of the trades.

### 3.6.12 Examples of double auctions

In this section, we give several examples of auctions with specific programming problems.

**Single-sided auction**

Since we have already gone through a double auction, we will start with this as a baseline. In a single-sided continuous and simultaneous auction, we will allow certain subjects or certain groups of subjects to do things that other subjects or groups of subjects are not allowed to do.

To do so, we first program an entire double auction, i.e. we program every box that will be passed through by any subject or group of subjects. Then we define and implement the conditions that allow only the desired subjects or group of subjects to see a certain box.

Conceptually, let's go through the treatment step by step:

1.  To allow only one group of traders to post offers, you can hide the corresponding boxes, i.e. for the players who cannot make offers you hide the "make offer" box. To do so, there is a display condition in the box dialog to dynamically show and hide boxes. A box is shown if the condition in the display condition is empty or if the condition is satisfied. The condition is calculated in the subjects' table. Whenever data in a subjects table changes, z-Leaf checks whether a box must be shown or hidden.

    In your example, the contract creation box would only appear for the group of traders who is allowed to post offers. Therefore you put the condition for the group posting offers into the display condition dialog, e.g. `IsSeller == 1` so that only the sellers are allowed to post offers.

    All the groups will be able to view the contract list box if there is no restriction in the display condition dialog.

2.  Next, you have to hide the sell button for the group that is not allowed to post offers. To do this, we create two different contract list boxes – one with a sell button and one without a sell button. Now we need to assign the boxes to the two groups of subjects, sellers and buyers, using the display condition. In the box with the button we write in the display condition `IsSeller==1`. In the box without the sell button we write in the display condition `IsBuyer==1`. This way, only the group that is allowed to post offers (the sellers) will be able to press the sell button.

**How can sellers offer different units at different marginal costs?**

In this double auction, the sellers produce increasing amounts of a good at decreasing marginal costs. Therefore we have to define the sellers' decreasing marginal costs. To do so, we will sort the sellers' marginal costs in decreasing cost ranks and enter the goods into a table. The buyers can then accept different goods units at decreasing marginal costs. To do this, let's go through the treatment step by step:

1.  We first need to create a new table in the Background with Treatment → New Table for the marginal costs that contains the following variables:

    – *Seller*: subject ID

    – *Cost*: costs

    – *CostRank*: rank of the firms' marginal costs

    – *Sold*: 1, if sold, 0 otherwise

2. Then in a program for a subjects table in the Background, we create a new record "marginalcosts" (using a new statement) for all firms to calculate their costs.

The program looks as following:

```
if ( Type == 1 ) {
    marginalcosts.new {
        Firm = :Subject;
        Cost = 10;
        CostRank = 1;
        Sold = 0;
    }
    marginalcosts.new {
        Firm = :Subject;
        Cost = 20;
        CostRank = 2;
        Sold = 0;
    }
}
```

This way, we have assigned an increasing cost ranking for the firms' costs manually. Of course, the rank could also be determinded using a program.

Additionally to the double auction as described, please consider the following important aspect in this double auction. Consider a firm that sells its offer. In the button "Sell" you write a program into the subjects table that deletes every last minimal cost rank whenever the variable Sold is set to 1.

The program in the subjects table is the following:

```
LowestCostRank = marginalcosts.minimum ( Firm == :Subject & Sold == 0, CostRank );
marginalcosts.do {
    if (CostRank == :LowestCostRank & Firm == :Subject & Sold == 0) {
        Sold = 1 ;
    }
}
```

### 3.7 Posted offer markets

In a posted offer market, one market side, say the sellers, makes offers. The other market side – in our case the buyers – can then, one after the other, select one of the offers. In z-Tree, this seller stage is easily implemented with a normal (non-auction) stage in which the sellers make their offers in a contract creation box.

The buyers select the contract in a contract list box. Now, to allow buyers to act sequentially, we use the Number of subjects in Stage option in the stage dialog. By setting it to At most one per group in stage, only one subject per group may enter the stage and therefore, the subjects enter the stage one after the other.

If nothing is specified, the order in which they enter is random. If the variable `Priority` is set, the subjects with lowest value (best rank), enter first.

If a subject has to go through several stages before the next subject may enter the sequence of stages, the second and all following stages must have the … and in previous stage(s) option set.

### 3.7.1 Example for a posted offer market

In a posted-offer market, sellers choose prices independently, and buyers cannot bargain individually, but act sequentially. The first-come-first-serve principle applies for the buyers.

To program a posted-offer market, we define the stages for the sellers and the buyers separately. Therefore at the beginning of a stage, we put into the subjects table the condition `Participate`, e.g. for the sellers: `Participate = if ( IsSeller==1, 1 0 );`.

Now in the buyers' stage, we have to make the buyers act sequentially. We can do this by applying specific options in the Stage dialog. To open the Stage dialog, please double click the buyers' stage. In our example, this is the stage Buy offer.

In the Stage dialog of the stage Buy offer, we use the option Number of subjects in Stage. If we do not specify this option, the subjects enter the stage all together. For a posted-offer market however, we set the option to At most one per group in stage so that only one subject per group enters the stage and therefore, the subjects enter the stage one after the other. The order of the subjects' entries is set randomly.

In this option field, there is also the option … and in previous stage(s). This option has to be set if a subject has to go through a sequence of stages before the next subject may enter the sequence of stages. In our example, this option is not selected.

Besides this, another possibility is to let the subjects enter a stage according to their rank, i.e. the subject with the lowest rank enters the stage first. This way, the order is not random anymore. To do this, we define a variable `Priority` in the subjects table and the variable for the subjects' rank in the globals table.

1.  We write the following program in the globals table at the beginning of the buyers' stage:

```
subjects.do {
   SubjectR = :Subject + random();
}
subjects.do {
   SubjectsRank = subjects.count ( SubjectR >= :SubjectR );
 Priority = SubjectsRank;
}
```

2.  And in the subjects table at the beginning of the buyers' stage as well, we write:

```
SubjectR = :Subject + random();
```

This way, the subject with the lowest rank enters the buyers' stage first.

Now, we also need to define the option Start. If the subjects are supposed to enter a stage sequentially, we have to set the option Start if possible, i.e. that a subject can only enter a stage if the previous subjects have passed through the stage (in other words, the subject waits his turn).

As a last specification in the stage dialog, we set the option Leave stage after timeout. In our example, we set it to If no input with a Timeout of 30 seconds. This means that the stage is terminated and the new stage started whenever the time of 30 seconds has expired – regardless of whether there is an input or not. We have now programmed a posted offer market and finish the treatment by displaying the profits made by sellers and buyers.

### 3.8 Clock auctions and deferred actions

### 3.8.1 The Dutch auction

In a Dutch auction, when a good is sold, there is a clock showing a price. This price decreases with time. As soon as one person decides to accept, she gets the product at the current price.

To implement this institution in z-Tree, we need a flexible clock. We must be able to define the steps of the price decrease as well as the time interval between price decreases. This can be done with a later statement. If you start with a price of 1000 and every 3 seconds you want to decrease the price by 10, you write the following program (for instance in the globals table):

```
Price = 1000;
later ( 3 ) repeat {
    Price = Price - 10;
}
```

The later statement has the following general form:

```
later( expression ) repeat { statements }
```

1. The expression is evaluated. Let $t$ be the value of the expression.

2. If $t$ is smaller than zero nothing more happens.

3. If $t$ is at least 0, then after $t$ seconds, the statements are executed and then we go back to 1.

There is a second form of the later statement for the situation where we want the statements to be executed at most once. In this case, we write:

```
later( expression ) do { statements }
```

Also, in this case the expression is evaluated. If the value $t$ is negative, nothing happens. If the value is greater than or equal to zero, then after the corresponding number of seconds pass, the statements in the curly brackets are executed.

### 3.8.2 Leaving a stage

Suppose you conduct a dutch auction with multiple groups. You would like the group to proceed to the next stage as soon as the good is sold. You can achieve this as follows:

First, you initialize a variable `Accepted` in the subjects table to 0. Then, you create the `later` command as above. You show the price in a standard box. In this box there is a button and in this button you put the following program.

```
if( sum( same( Group ), Accepted ) == 0 ) { // no other player was quicker
    Accepted = 1;
    subjects.do {
        if ( same( Group ) ) {
            LeaveStage = 1;
        }
    }
}
```

The variable `LeaveStage` allows you to force subjects to leave a stage. In the program above, all subjects in the group of the subject who accepted the price immediately leave the stage.

### 3.8.3 Auction trial periods with simulate data

You plan to conduct an experiment with a double auction. Because the user interface will be rather difficult, subjects should be given some time to learn how it works. This is best achieved with some trial periods. A problem with trial periods is that they allow interaction between the subjects and uncontrolled learning before the actual experiment starts. To allow for controlled learning, you can simulate the entries of the other subjects in the trial period auctions. This can be done with a sequence of `later` statements.

### 3.8.4 Double auction with an external shock

Suppose you want to change the economic environment during a double auction. Let *P* be the parameter you want to change from 100 to 50 after a minute. You write the following program:

```
P = 100;
later ( 60 ) do {
    P = 50;
}
```

### 3.8.5 Exit in an ascending clock auction

In an ascending clock auction, the subjects are able to press a button whenever they wish to exit the auction. The winner is determined when the second to last subject has exited. For an auction with ascending prices, we define an "Exit" button for subjects to exit the auction whenever they wish to do so. Then we need to write a program in the subjects table into the Exit button: As soon as a subject has

pressed the button, the button disappears (`ShowExitButton = 0;`). Furthermore, we need to define what happens if there is only one subject left in a group that has not pressed the button yet.

If there is only one subject left, we want the whole group to leave the stage because the auction then has ended. The corresponding condition is the following:

```
if ( count ( same ( Group ) & Out == 0 ) <= 1 ) {
    subjects.do {
        if ( same ( Group ) ) {
            LeaveStage = 1;
        }
    }
}
```

The group now leaves the auction stage and enters the result stage in which the subjects are informed about the winner's price and whether they themselves are the winners or not.

Do not forget to put a checker into the Exit button. This checker contains the following condition:

```
count ( same ( Group ) & Out == 0 ) >= 2
```

If this condition is not satisfied (anymore), there is only one subject left that has not pressed the "Exit" button yet. This subject has won the clock auction.

### 3.8.6 Displaying time

Stage time counts the starting time from when the time gets control. This causes some delay. Similarly, the repetition of the later command can be delayed due to activities in z-Tree. If you want to display a clock that is independent of these delays, you can combine gettime() with the later command as shown in the following example. Note that the delay can be longer than the time between two ticks. This problem can be handles in different ways. For example, if a time has to be displayed, it makes sense to omit steps as necessary to catch up. This is done in the while loop.

```
PreferredRefreshRate = 1; // 1 second

StartTime = gettime();
NextTime = StartTime + PreferredRefreshRate;
TickTime = NextTime - StartTime;
later ( TickTime ) repeat {
  // any stuff
  subjects.do{
    ShownTime = gettime();
  }

  repeat {
    StartTime = gettime();
    NextTime = NextTime + PreferredRefreshRate;
    TickTime = NextTime - StartTime;
  } while ( TickTime <=0 ) ;
}
```

### 3.9 Free form communication and strings

Since z-Tree version 3.4, strings can be used in every item. Thus, a chat function can also be implemented using standard elements. In this section, we will first demonstrate how to implement a chat treatment using a chat box. It is useful because it integrates all the functions necessary for a chat. In addition, it allows messages to be finished with the return key, and it provides an output format that is particularly useful for chats.

### 3.9.1 The chat box

The chat box allows controlled communication between subjects, and it uses the contracts table or a user-defined table. Subjects see a text entry field, and when they press the enter key on the keyboard, the text is stored as a new entry in the contracts (or user-defined) table. In this example, we will allow two subjects to chat with each other, and a third subject to observe the conversation. We include the observer to show how chat text can be treated as both input and as output. First, in the Background the number of subjects should be changed to a multiple of 3 (it is best to test it with several groups to make sure it works). Next we will initialize several variables in the Background. In the subjects table, we assign groups and player ID, so that there are three subjects per group, and each one has a player ID of either 1, 2, or 3. We use this variable to designate the observer: in our case subjects with a player ID of 1 are observers. The code in the subjects table can be seen in the first program in Figure 3.35, "Variable initialization in the chat example". Next, we will initialize variables in the contracts table. The variables in the contracts table Owner, ID, and Groupie are simply the corollaries of the subjects table variables Subject, PlayerID, and Group. We initialize them in order to use them in the display condition of the chat box (which we will explain next).
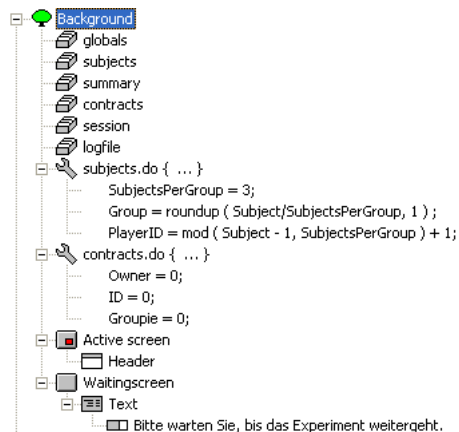
Figure 3.35. Variable initialization in the chat example

In the Active screen, select Treatment → New Box → New Chat Box…. There is a field called Display condition as well as a field called Condition. The former determines whether the box itself will be displayed or not. The latter determines which rows of the contracts table (or user-defined table) will be shown in the chat output. You should insert two chat boxes, one for the chatters and one for the observer. Figure 3.36, "Chat box for the chatters" shows the chat box for the subjects who are allowed to chat, Figure 3.37, "Chat box for the observer" shows the chat box for the observers. The display condition for the chatting subject is `PlayerID != 1`, for the observer it is `PlayerID == 1`. The table is set by default to contracts, which we will use. In the chat box for chatters, in the area Input var. insert the name of the variable which you would like to represent the chat messages (in our case, `Words`). In the Condition field, specify which records of the contracts table should be shown. In our case, we wish for subjects in one group not to be able to see the other groups' chat messages. In the Condition field write `Group == :Group` so that only those records from the contracts table with a `Group` value equal to the `Group` variable from the subjects table are shown. The variable `Words` is automatically entered as a new row to the contracts table every time the enter key is pressed.

Figure 3.36. Chat box for the chatters



Figure 3.37. Chat box for the observer

However, the information about group and player ID in the contracts table is not included automatically, so we can include a program within the chat box to enter this information. In the chat box, enter a new program and change the table to contracts. Enter the code shown below to keep track of group, subject ID, and player number within the group.

```
Subject = :Subject;
Group = :Group;
ID = :PlayerID;
```

Each time the enter key is pressed, the new row of the contracts table will be modified to include this information. So in addition to using the group variable in the Condition field, we can use the player ID variable in the output text. This is where you can enter what text variable appears. The syntax `<><|-1>` appears by default in the Output text field. If you leave this blank and there is an input variable, the input variable will automatically be shown as output. If there is no input (as it is the case for the observer), you must include the variable. In the Output text field of the observer's chat box, writing `<><Words|-1>` will allow the observer to see the chats of the members of her group. As in any chat situation, it is useful to know who is saying what. So, we will imbed the player ID in the output text. The Output text field should then appear as

```
<>Player <ID|1>: <Words|-1>
```

which allows the participants to see the player ID in front of each chat message. The `<>` symbols are used when a variable included within text will occur. In the code shown above, we include two variables: `ID` and `Words`. The text "Player" is included before each chat message, but is not a variable, so does not have to appear within the '`<>`'. The variable `ID` is a numerical variable and we want it to be represented as such, so after the '`|`', we use the number 1 to indicate that the layout is numeric in whole numbers. Since `Words` is a text variable, the layout code is -1. With this our chat environment is now finished.

### 3.9.2 String operators and functions

In z-Tree version 3.4, strings have been implemented as variables, meaning it is possible to modify or analyze text such as the chat messages we have just discussed. As an example, we will make a program that counts the number of characters it takes until a smiley emoticon first occurs in a written text. In addition to the chat box, text can be written in a standard box. Here, however, the return key does not function to "send" a chat message. In the first stage, insert an input item into the standard box.
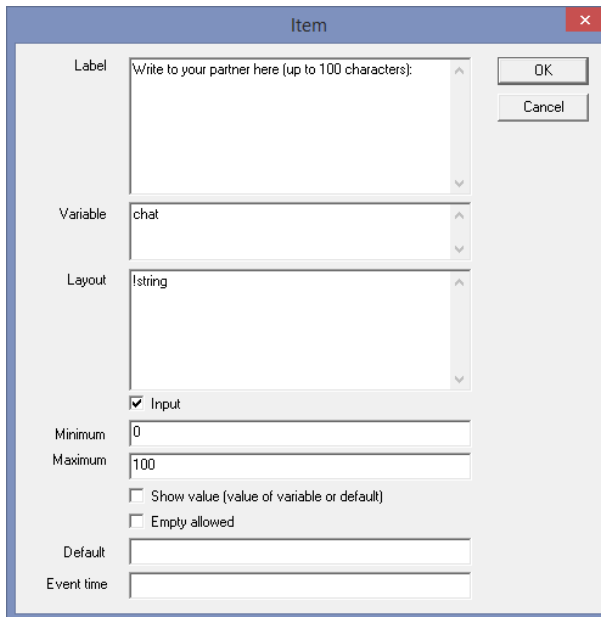
Figure 3.38. Input item for a string variable and what appears on the subject's screen



Figure 3.39. String input as it appears on the client's screen

Figure 3.38, "Input item for a string variable and what appears on the subject's screen" shows what the input item should look like. The minimum and maximum values determine the number of characters a subject is allowed to write, and the layout option `!string` allows a subject to enter text. An example of what a subject may then write in can be seen on Figure 3.39, "String input as it appears on the client's screen".

We next insert the program

```
smileyPlace = pos(chat, ":)", 1);
```

into the beginning of the second stage. This program counts the number of characters after which the string ":)" occurs within the string variable `chat`, and begins counting at the character number 1. With an output item (Figure 3.40, "Output the position where the smiley emoticon was found") we output this information to the subject (Figure 3.41, "String output as it appears on the client's screen").
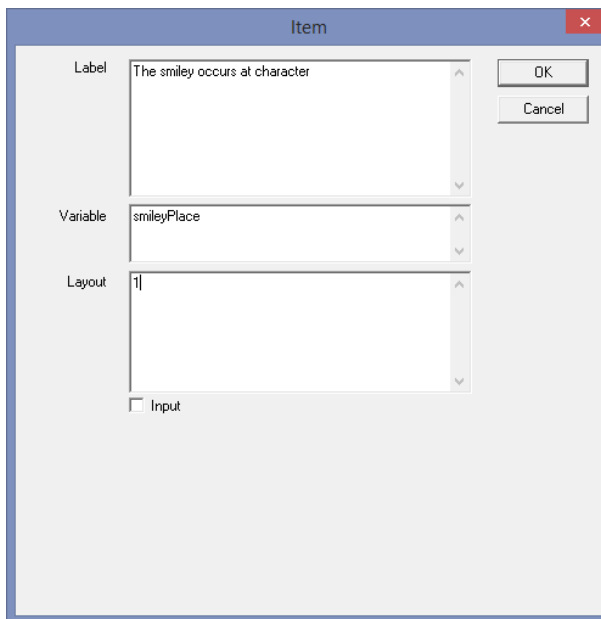
Figure 3.40. Output the position where the smiley emoticon was found
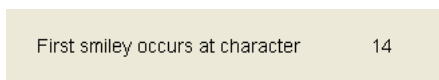


Figure 3.41. String output as it appears on the client's screen

## 3.10 More programming: if, loops, arrays, and data types

### 3.10.1 Conditional execution of statements

If you wish to carry out something only under certain conditions, you can utilize the if-statement. It may be utilized in the following forms:

```
if ( condition ) { true_statements }
if ( condition ) { true_statements } else { false_statements }
```

The condition is evaluated first. If it is TRUE, the instructions in the `true_statements` are carried out. If the condition is FALSE, nothing is done in the first case, and in the second case the instructions in the `false_statements` are carried out.

**Example**

```
if ( type == FIRM ) {
    Profit = v * e - w;
    othersProfit = w - c;
}
else { // type == WORKER
    Profit = w - c;
    othersProfit = v * e - w;
}
```

You may also carry out conditional calculations by using the *if function*. However, carrying out many calculations in this way requires constant repetition of the condition. This is inefficient and can become confusing.

> The if-statement makes it possible to calculate a variable for some records and not for others. This can result in undefined cells. It is a good practice to initialize variables at the beginning of the treatment.

Consider a situation where you have 4 types of players and for the different types you have different profit functions. You can program this with a nested if statement as follows:

```
if ( type == 1 ) {
    // first calculation
}
else {
    if ( type == 2 ) {
        // second calculation
    }
    else {
        if ( type == 3 ) {
            // third calculation
        }
        else {
            // fourth calculation
        }
    }
}
```

It is not easy to keep track of all the necessary closing brackets. With `elsif`, there is an easier way to express the above program (there is only one 'e' in elsif):

```
if ( type == 1 ) {
    // first calculation
}
elsif ( type == 2 ) {
    // second calculation
}
elsif ( type == 3 ) {
    // third calculation
}
else {
    // fourth calculation
}
```

### 3.10.2 Loops: while, repeat and for

There are three forms of loops in z-Tree. The first is:

```
while( condition ) { statements }
```

The condition is evaluated and as long as the condition returns TRUE, the statements are executed and the condition is reevaluated. The second is:

```
repeat { statements } while ( condition );
```

The statements are executed, and then the condition is evaluated. As long as the condition returns TRUE, the statements are executed again and the condition is reevaluated. With the `repeat` statement, the statements between the curly brackets are executed at least once.

Loops are frequent sources of program bugs. For example, if the condition in a loop never returns FALSE, then the loop runs forever. You can leave a loop with **Ctrl+Alt+F5**. This, however, results in undefined results in z-Tree and should only be used when testing a treatment (by the time of conducting a session there should be no more infinite loops, since they are errors).

A third way to program loops is by using *for*. This form of a loop is available in z-tree 4.0 and higher. It replaces the iterator, which is no longer available. Below, we will show how to replace iterators by for loops. A for loop iterates a variable from one value to another. For example, you want to add the numbers between 3 and 7, then you can write the following program.

```
Sum = 0;
for ( i, 3, 7 ) {
  Sum = Sum +i;
}
```

The program in the curly brackets is executes sequentially setting the value of i to 3, 4, 5, 6, and 7. Specifically, the variable i is initialized with the value of 3. Then it is incremented in each step by 1 as long as it does not exceed 7. In principle, you can modify the value of i inside of the loop. It is usually not a good idea because it makes the program much less readable. After the loop, i has a value of 7.

There are four forms of for loops:

```
for( varname )          // runs from 1 to number of subjects
for( varname, n )       // runs from 1 to n
for( varname, x, y )    // runs from x to y
for( varname, x, y, d ) // runs from x to y with steps of d.
```

If the step is negative, then the for loop counts down. It is possible to change the loop variable, the upper limit, or the step size inside of the loop. This will change the terminating condition of the loop. We recommend not to use this feature because readers of your program might get confused.

**Example**

```
// sum of all odd numbers between 0 and 100
Sum = 0;
for (i, 1, 100, 2 ) {
  Sum = Sum + i;
}
```

**Replacing iterators by for loops**

Iterator are like tables and they generate a new scope environment. Thus, you have to remove the some scope operators if you replace the iterators by a for loop. Here is the above example with an iterator:

```
// sum of all odd numbers between 0 and 100
Sum = 0;
iterator (i, 1, 100, 2 ).do {
  :Sum = :Sum + i;
}
```

Thus, you have to change two things: Delete `.do`, and remove the scope operators because the for loop does not create a new scope.

### 3.10.3 Boolean expressions and variables - and data types in general

Since version 4.0, it is possible to use conditions in variables. Conditions are also called boolean expressions. This means that you can write the following program.

```
X = 1;
Y = 2;
B = X<=Y;
Smaller = if(B,X,Y);
```

In z-Tree, the following data types are possible : `real`, i.e., numbers, `string`, `boolean`, and `color`. The latter will be explained in Section Section 3.11, "Introduction to graphics". The programming language of z-Tree is strongly typed. This means that a variable cannot contain content of different types. This is demonstrated in the following program.

```
X = 1;
X = "1"; // error message "Wrong type X"
X = 1<=0; // error message "Wrong type X"
```

### 3.10.4 Arrays - variables with an index

Arrays are variables with an index. For example, we can create discrete demand and supply schedule using and array. We define an arrays for the marginal costs and the redemption values in the subjects table in the Background. To do so, you first write the array:

```
array Rvalue [4];
array Cost [4];
```

And then you write the values for each index. In our example:

```
RValue [1] = 400;
RValue [2] = 300;
RValue [3] = 200;
RValue [4] = 200;
Cost [1] = 20;
Cost [2] = 120;
Cost [3] = 220;
Cost [4] = 220;
```

Then you need to define the index of a trade to assign the corresponding values for the costs and the redemption values:

```
// initialize
NumTrades = 0;
```

```
// a trade occurs
NumTrades = NumTrades +1;
TradeRvalue = RValue[NumTrades];
TradeCost = Cost[NumTrades];
```

By default, the elements of an array are numbers (reals). You can also use arrays of other types. In this case, you have to specify the type. In addition, you can define arrays with the same specifications as the for loop.

```
real array A[10]; // as array A[10];
boolean array BA[3,6]; // boolean BA[3], BA[4], BA[5], BA[6],
string array StrA[3,6,2]; // string StrA[3], StrA[5]
color array C[2];   // colors see below in the graphics section
```

### 3.11 Introduction to graphics

### 3.11.1 Graphic display

In this section we will create an on-screen image: a smiley face. To display graphics in z-Tree, you use a plot box. The plot box sets up a coordinate system so that you can position different types of shapes and other elements on the screen.
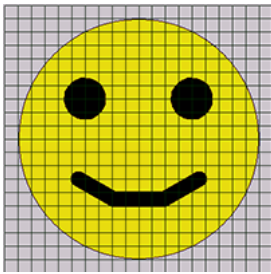
Figure 3.42. An on-screen smiley face
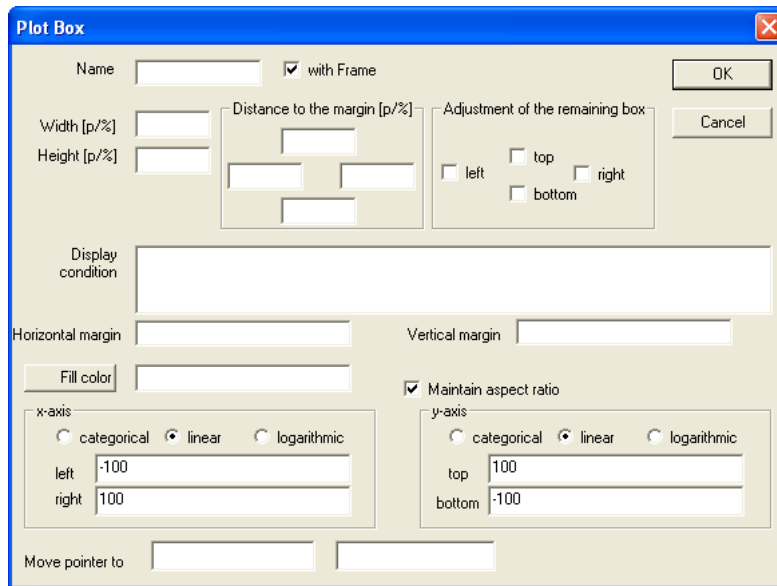


Figure 3.43. The canvas for a smiley face

First you should insert a new plot box into the Active screen of a stage (Figure 3.43, "The canvas for a smiley face"). Make sure to check the box Maintain aspect ratio if you wish for your images not to appear distorted when you display them on different types of computer screens. The smiley face shown in Figure 3.42, "An on-screen smiley face" is made up of 3 different graphic elements: a pie (the yellow circle), two points (the eyes), and 3 lines (the mouth). In the plot box, go to the Treatment → Graphics menu and insert what will be the first element of the face: a "new pie". You can choose the angle size of the pie (in our case 360° to make a full circle), the location of the apex of the pie (or center of the circle), as well as the radius (you can choose different lengths in x and y dimensions if you want to make an elliptical shape). Click on Fill color to choose from a menu of colors if you do not wish to write in the rgb code directly.
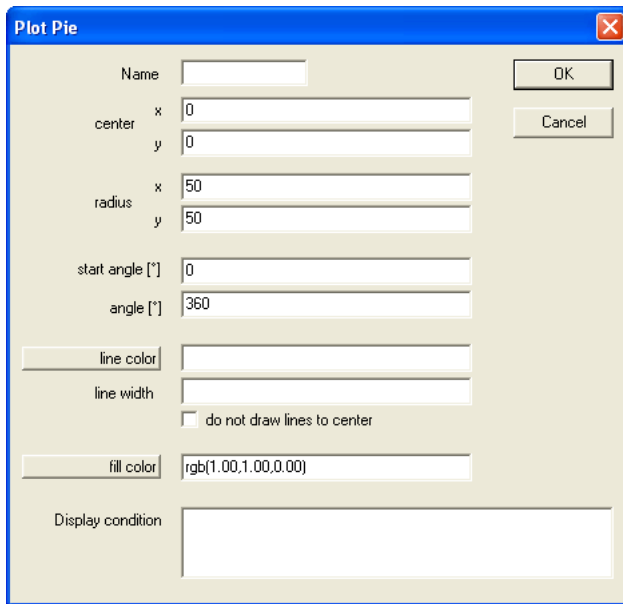
Figure 3.44. Plot pie: Smiley's face

Next, insert two points. Increase the Number of vertices so that the eye appears as a circle rather than a polygon (stars are also possible). Figure 3.45, "Plot point: Smiley's right eye" shows the location of the right eye; the left eye should be identical but located at the point (-20, 20).
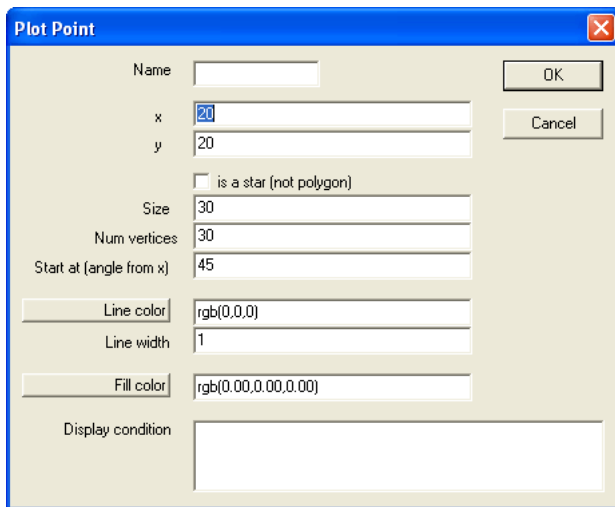


Figure 3.45. Plot point: Smiley's right eye

The next item to insert is a line. The first line to insert is shown in Figure 3.46, "Plot line: Smiley's mouth (middle part)", and is the bottom of the mouth. Two additional lines, going from the point (-20,-30) to (-30,-20) and from point (20,-30) to (30,-20) complete the mouth.
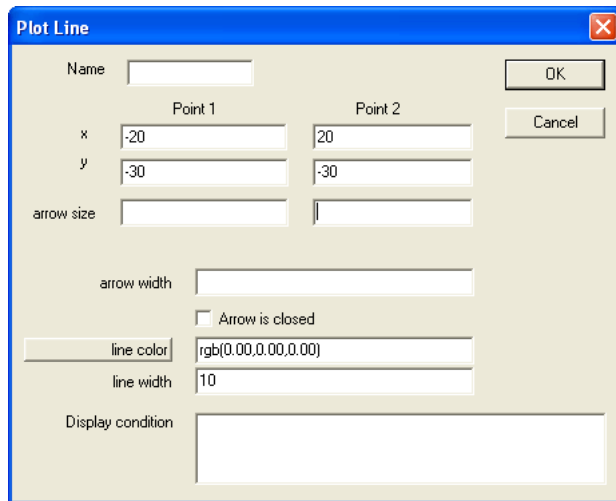
Figure 3.46. Plot line: Smiley's mouth (middle part)

Your smiley is now finished!

### 3.11.2 Plot graphs

Plot graphs are used to create graphs, charts, scatter plots. A Plot graph sets up a sequence of records in a table. For each record, graphical elements can be drawn. The parameters of the graphical elements are evaluated in the corresponding record. The scope environment is the own record in the subjects table and the globals table - as in a contract list box. Plot points can be connected. Plot graphs can also be nested. In this case the scope environments of the graphical element contain all the records in which the element is contained.

We show how to create a bar chart as in the following figure. All bars are in addition connected with a line.
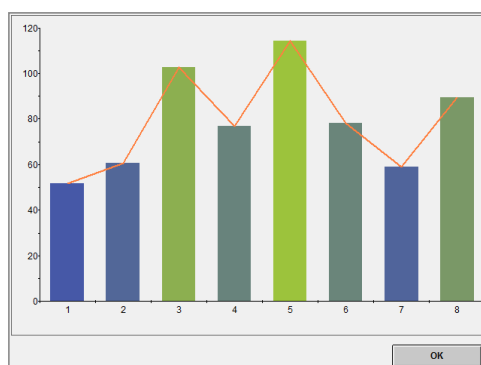


Figure 3.47. Screenshot of plot bars.

Below, we show the stage tree of the treatment. In the program we fill 8 records into the contracts table. There we specify the X coordinate and Value, which will define the height of the bar. In addition, we define the color of the bar. We make the color depend on the height.
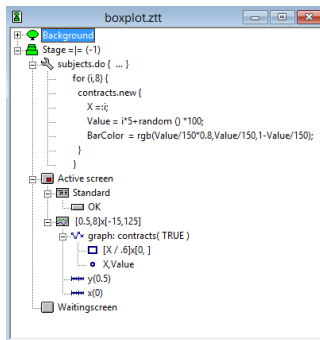
Figure 3.48. Stage tree of the plot bars treatment.

We add a small standard box, which is positioned at the bottom of the screen, in order to be able to finish the stage easily. The basis of the bar chart is the plot graph. It defines the data that will be displayed. All records will be displayed because the condition is TRUE. Because we define line color and width, all points in the plots graph will be connected in this style.
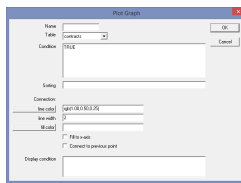


Figure 3.49. Plot graph of the plot bars treatment.

In the plot rect, we specify where and how to draw it.
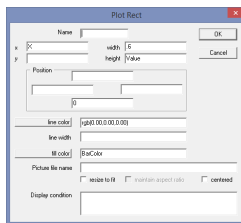


Figure 3.50. Plot rect of the plot bars treatment.

For the plot point, we only specify where to draw it. Because line and fill options are empty, the point will not be drawn. But because a line is specified in the plot graph, the points will be connected.
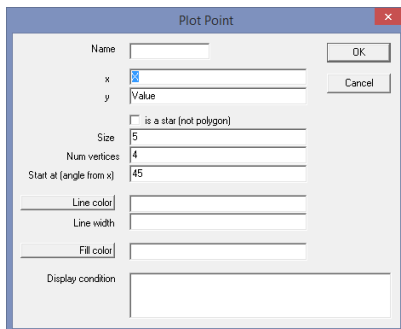


Figure 3.51. Plot point of the plot bars treatment.

**Exercises**

– Create individual bar charts for the subjects.

– Create a scatter plot diagram.

– Create a bar chart.

– Create forest consisting of similar trees.

### 3.11.3 Color operations (version>=4.0)

Colors can also be assigned to variables and there are couple of color operations. You can define colors using rgb and hls (hue, lightness, saturation), which expect value between 0 and 1, or rgb255 and hls255, which expect numbers between 0 an 255. You can get the red, green, and blue component using getred(c), or getred255(c). Finally, there are two mixing functions.

– mix(c1, c2) returns a mixed color that respects the visual impression.

– mixlinear(c1, c2) returns a color with the component wise averages.

### 3.11.4 Interactive graphics

In this section we will go through several examples to learn how to use interactive graphics. There are many possibilities for modification of graphics in addition to the typical mouse operations such as clicking and dragging objects. To make graphics interactive, you use the item *plot input* inside the plot box. There are three possible actions that can occur when using a plot input, which are *new*, *select*, and *drag*.

**Action New**

To illustrate how the option New works, we will use the smiley face from the last example. In this example, the left eye of the smiley will reappear wherever you click on the screen. To keep track of the coordinates where the eye should be located, we will give its x and y coordinates variable names. In the Background, insert a new program in the subjects table, creating a variable called LeftEyeX and LeftEyeY (equal to -20 and 20, respectively). Next replace the x and y coordinates of the left eye point with the new variable names (Figure 3.52, "Smiley's left eye").
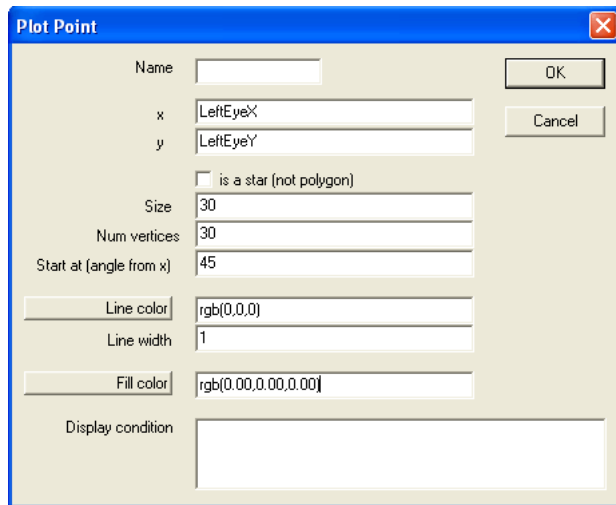
Figure 3.52. Smiley's left eye

Next insert a plot input into the plot box. Make sure that the plot input is located inside the plot box, and not inside any graphic item (a line or point, for instance). Plot inputs can be put inside graphic items, but this only make sense for the action options Select and Drag, rather than New. As shown on Figure 3.54, "Stage tree of the example for action new", the plot input item will trigger a program to run upon a left click of the mouse. In the area New (Action New) of the plot input, select the table subjects, and enter the names of the variables which will be updated when a left click on the screen occurs.
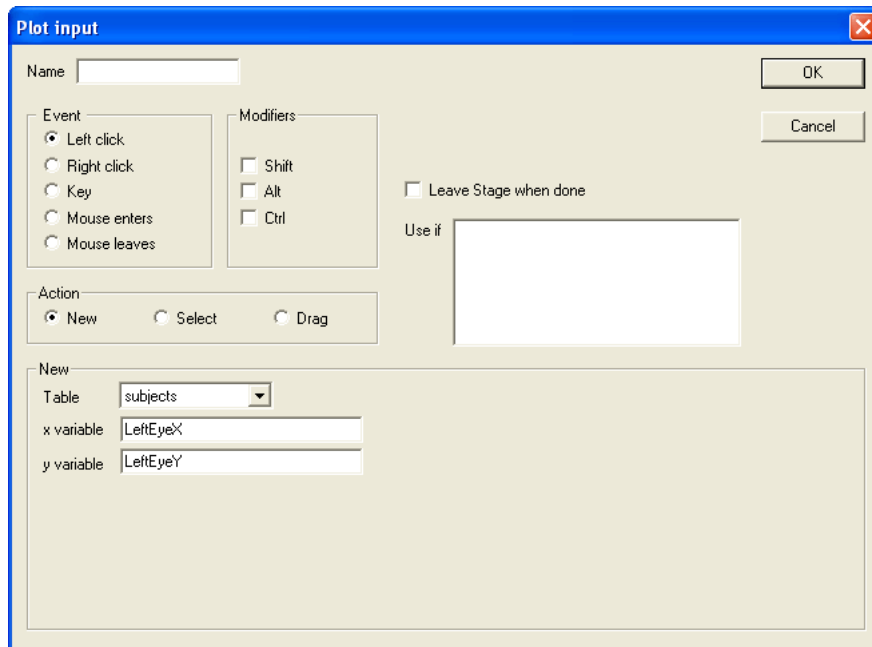


Figure 3.53. Record mouse clicks with plot input

When a click occurs on the screen, the variables LeftEyeX and LeftEyeY in the subjects table will be overwritten with the location where the mouse click was released. Since the subjects table has a fixed number of rows, information about previous clicks on the screen will be lost. To save this information, you can include a program under the plot input to add a record to the contracts table (or a custom table) every time a subject clicks on the screen.
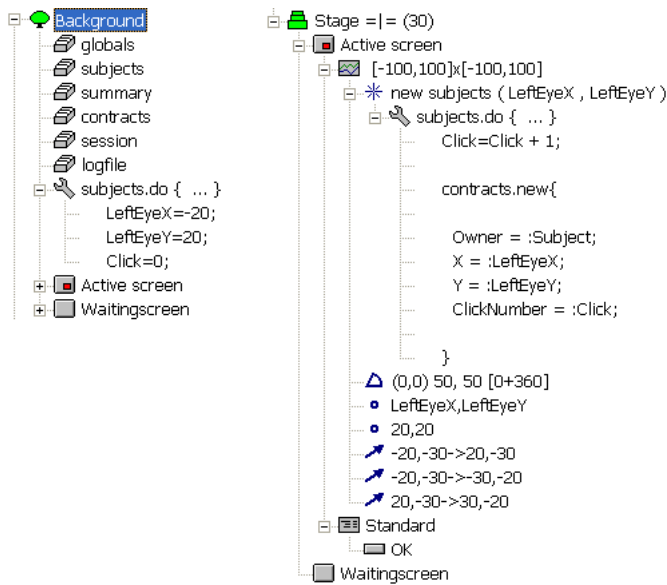
Figure 3.54. Stage tree of the example for action new

**Action Select**

As an example for the usage of the option Select we will program a treatment in which clicking on a rectangle triggers a program which changes the color of the rectangle. In the Background (see Figure 3.55, "Plot rect of the example for action select") introduce a variable called Selected to manipulate the color of the rectangle. As in the smiley example, we will need a plot box. In the plot box, we will select the graphic Treatment → Graphics → New Rect…. In the area fill color, include the variable Selected in any of the three color code areas.
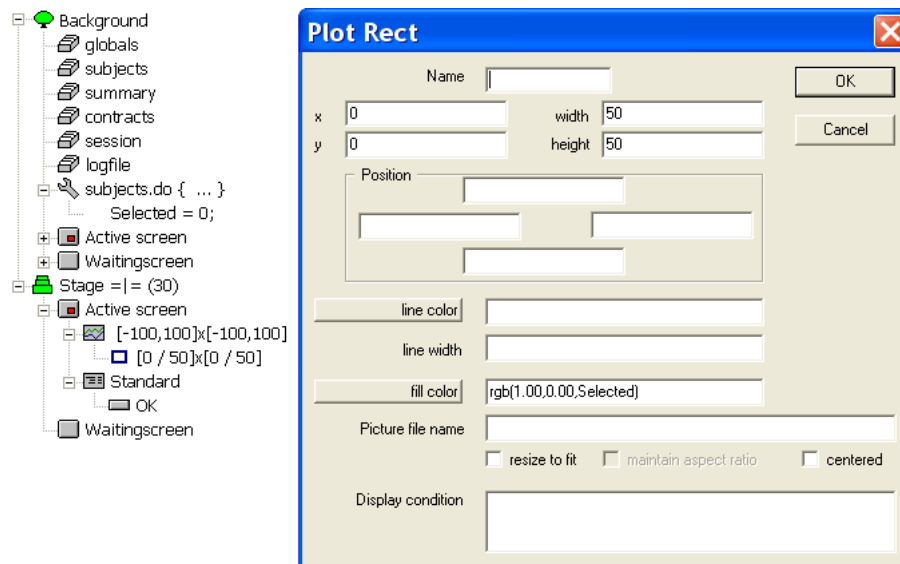


Figure 3.55. Plot rect of the example for action select

Next insert a plot input into the rectangle. Make sure the plot input is within the rectangle. In the Action area, New is selected by default, therefore change it to Select. With the *select* option, it is not necessary

to fill any information in for the x and y variables. Next, insert a program within the plot input item. The program within the subjects table should read

```
Selected = 1 - Selected;
```

which will make the variable alternate between values of one and zero every time the rectangle is clicked.
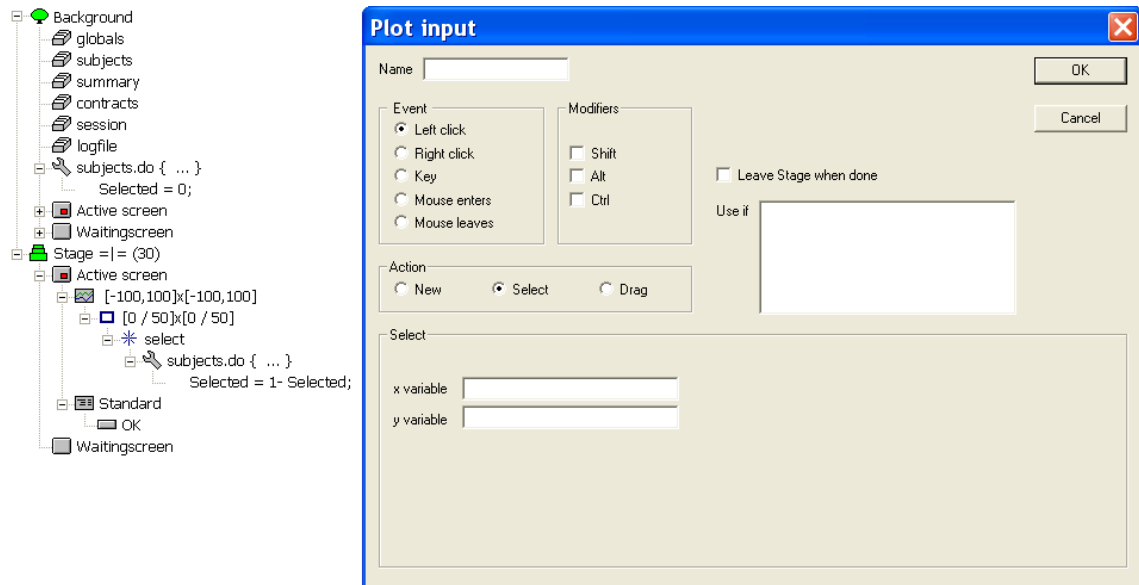


Figure 3.56. Plot input of the example for action select

**Action Drag**

The last example is the *drag* option. To illustrate the drag function, we will program a scrollbar. We will program a scrollbar in order to show how restrictions can be implemented (in this case, that the object can only be moved along the x dimension and not the y dimension). First, we will set some variable values to define the scrollbar limits as well as the scrollbar size and the initial location of the scrollbar. In a new program in the background, set the left bound to -75 and the right bound to 75 (called `Left-Bound` and `RightBound` in our example). Next, set the initial x location to 0, and the rectangle height and width to 10 and 10 respectively. Do this in the Background too, since we will need to refer to these values as variables later on. You can see these programs in Figure 3.57, "Slider example". Next, insert a plot box in the Active screen of a new stage. In the plot box should go the three lines and a rectangle. One horizontal line should go from the left bound to the right bound, and two vertical lines should bound the horizontal one. For the rectangle coordinates, set x to the variable x and y to zero. In the rectangle, insert a plot input item, and under Action click Drag. Several empty fields will appear. p0, p, and p' are the points which will define how the plot item will move in relation to the mouse. The *point p0* is the point where you first click the shape. The *point p* is where the mouse is currently moved, and *p'* is where the object should be moved to while dragging. This means that the object is moved by *p'-p*. Note that you have to take into account that the object is not necessarily be picked at the center, so we need to keep track of the old center of the object. To modify the location of the object when the mouse is released, insert a program into the drag input item. With these elements, we can complete the scrollbar. Now we will go into detail about the plot input item and the program within it.
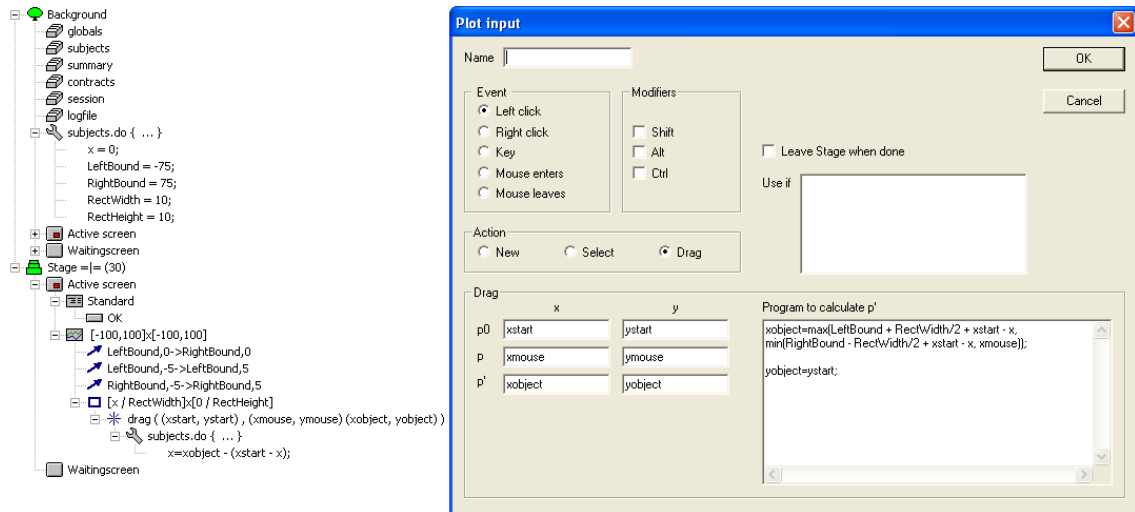
Figure 3.57. Slider example

In the areas `p0`, `p`, and `p'` we can assign variable names to the associated concepts. As shown in Figure 3.57, "Slider example", we name the initial location where the object is clicked `xstart` and `ystart`, the location of the mouse `xmouse` and `ymouse`, and the location of the object in relation to the mouse `xobject` and `yobject`. In the bottom right field of the plot input, you enter the program which determines where the object should be located in relation to the mouse. In Figure 3.57, "Slider example" there are two blocks of code; the first determines the x value and the second the y value. The y calculation is of course simpler, since the scrollbar should not move vertically at all. So, `yobject` (the y value of the object as it is being moved) should be equal to `ystart`. We do not need a program for the y value after the mouse is released, since the rectangle should remain located at zero in the y dimension. The x calculation is quite a bit more complex. There are several elements included. The first is the use of the *minimum* and *maximum* functions. The minimum function, with the format `min(a,b)`, returns the minimum of the two inputs a and b. The maximum function has the same format and returns the maximum of the two. In our example they are imbedded within one another; let us first look at the `min()` function. The code to calculate `xobject` includes the term `min(RightBound - RectWidth/2 + xstart - x,` `xmouse)`. This means that the rectangle should not appear beyond the scrollbar limit area. `xstart -` `x` is the distance between the center of the object and where it was clicked, and `RectWidth/2` is the distance from the center of the object to its edge. Since we want that the rectangle's edge should go up to the limit line but not beyond it, we allow it to be moved to the right bound, making sure that it is not moved farther than the amount of space between the right edge of the rectangle and where the rectangle was clicked. The maximum function is used for the left bound. It performs the corollary function of not allowing the scrollbar rectangle to move beyond the left bar.



Figure 3.58. Slider handle calculations

The last piece of code is found in the program below the drag item. It updates the value of x, subtracting the distance from the center of the object to the location where it was clicked. So, as shown in Figure 3.57, "Slider example", the program should read `x = xobject - (xstart - x)`.

### 3.12 Complex move structures

### 3.12.1 The start if option

It is possible to implement any kind of move structure in z-Tree. For more complex move structures, use the start option Start if…. We show how to use this option in a treatment with the following decision structure: There are three players, A, B and C. Players A and B move first, then player C moves. However, player C only finds out the move of player A. So, we want to allow player C to decide as soon as player A has decided. This would be reasonable for instance if the decision of player B were much more time consuming than the decision of player A.

Let `Type` be the variable that contains the type. Let `DecisionA` be the input variable of player A. Let `DecisionB` and `DecisionC` be the input variables of player B and C. Assume that the variable must be positive so that if we initialize the variables with `-1`, we know who has made her entry. We define 3 stages, stageA, stageB, and stageC. Additionally, we assign a type value of 1 to player A, 2 to player B, and 3 to player C. In stageA, we wait for all. In stageB, we start immediately. In stageC, we start if

```
count( Type == 1 & DecisionA > 0 ) == count( Type == 1 )
```

With this condition, the players C start when all players A have finished. With

```
count( same (Group) & Type == 1 & DecisionA > 0 ) == 1
```

player C starts when her player A has finished. Of course, we also have to set the `Participate` variable accordingly.

### 3.12.2 Turn information on and off

Boxes can be shown and hidden dynamically. There is a display condition option in the box. A box is shown if the condition is empty or if it equals TRUE. Whenever data changes, z-Leaf checks whether a box must be shown or hidden. You can use this feature for instance to control the access to information. Below is an example what a subject could see using this option. A subject clicks the button "Show instructions", and the reminder message on the right is shown. To return to the decision screen, a subject clicks "Return to decision".
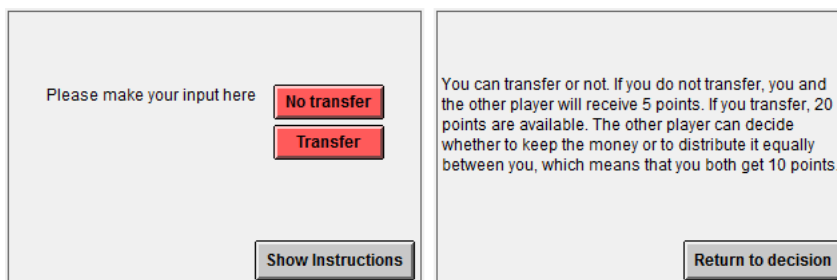


Figure 3.59. Turn information on and off with the display condition

The two grey buttons include programs which switch a variable between one and zero. This variable is used in the box display condition to control whether the information box is shown. Figure 3.60, "Display condition example" shows the stage tree of the example. In the decision box, the display condition is `ShowInfo == 0`, and in the instruction box the condition is `ShowInfo == 1`. The variable `showInfo` is modified when a subject clicks on one of the grey buttons. This happens by putting a program within the button, as shown in the figure as well.
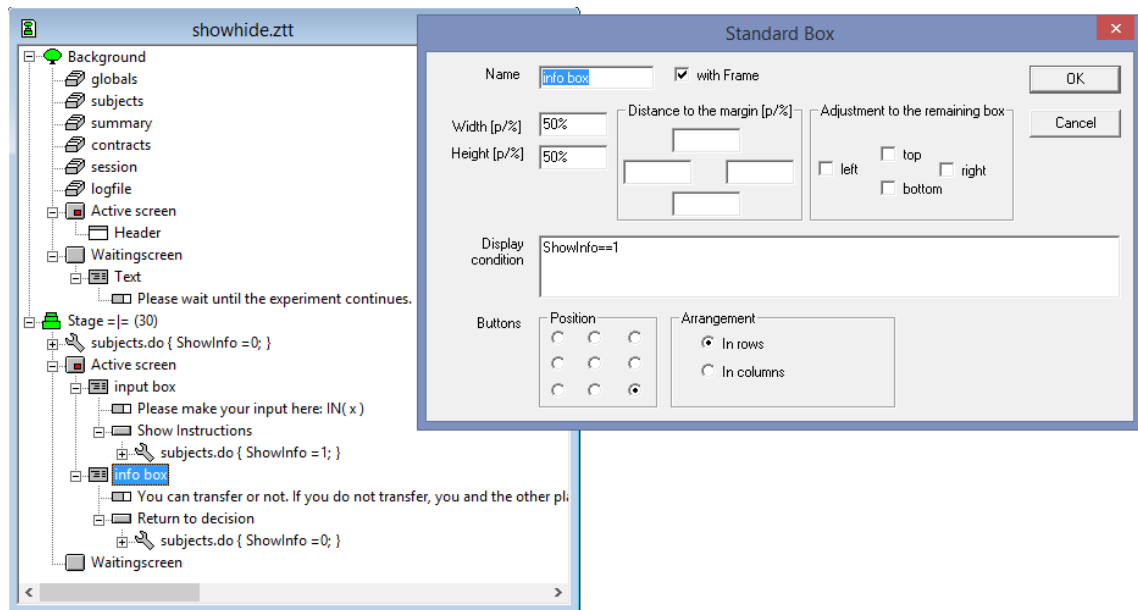


Figure 3.60. Display condition example

### 3.12.3 Moving back

Stages are always processed sequentially. It is not possible to return to previous stages. If you want to let subjects move back and forth, you can simulate stages using boxes. A good example are instructions organized in pages. You will want to allow subjects moving back and forth in the pages. For this you use a variable `CurrentPage` in the subjects table, which you initialize to 1. Then you create the pages, for examples in container boxes. Set the display condition of the first page to `CurrentPage==1`, the Display condition of the second page to `CurrentPage==2`, etc. You put NEXT and PREV buttons into the boxes, which contain the following programs:

```
// NEXT button
CurrentPage = CurrentPage + 1;
```

```
// PREV button
CurrentPage = CurrentPage - 1;
```

The programs change the variable `CurrentPage` and this turns the right page on and off.

### 3.12.4 Treatments of indefinite length

If you want to apply a random stopping rule, the treatment will have no definite length. This can be implemented in z-Tree with the variable `RepeatTreatment`. First, you define a one period treatment. At the end of the period you decide whether to continue the treatment or to stop. This decision is written into the variable `RepeatTreatment` in the globals table. If the variable is not defined, the treatment is terminated. This is the normal case. If you define the variable `RepeatTreatment`, it depends on the value of the variable whether the treatment is terminated or whether it is repeated once again. If the value is smaller than or equal to zero, the treatment stops. If the value is greater than zero, the treatment is repeated.

If the treatment is repeated then the period counter is incremented as if there were one treatment. This is implemented by modifying the number of (trial) periods. So, if you repeat a treatment with three periods, the first treatment has 3 periods and no trial periods. In the second run of the treatment, the treatment's number of periods is changed to 6 and the number of trial periods is changed to -3.

### 3.12.5 Example: Strategy method (using arrays)

Let us consider a treatment in which we wish to use the strategy method for a simple game (subjects make decisions for all possible cases that could occur, before actually knowing which one does). There are two types of players: A and B. The A players select a number a between 1 and 10, the B players select a number b for every possible a, i.e., they select b1, b2, b3,…b10. Let the profit be the absolute difference between the two numbers a and b. The formula for the calculation of profits looks something like this:

```
a = find( type == A, a);
Profit = if( a == 1, abs(a-b1),
         if( a == 2, abs(a-b2),
         if( a == 3, abs(a-b3), … ) ) );
```

This is very long-winded.

In such cases, it is wisest to define an array, i.e., an indexed variable. In an index set, every finite, equidistant subset of numbers is allowed. The index set needs to be defined by an array instruction before the first use of the array. The array instruction can have the following forms:

```
array arrayvar[ ]; // defines an array with indices from 1 to number of subjects
array arrayvar[ n ]; // defines an array with indices from 1 to n
array arrayvar[ x, y ]; // defines an array with indices from x to y
array arrayvar[ x, y, d ]; // defines an array with indices from x to y
                           // with distance d.
```

The access to array elements is carried out with `arrayvar[indexvalue`. The expression *indexvalue* is rounded to the nearest possible index and the corresponding value in the array is fetched or filled.

**Example**

```
array p[10, 20, 5];    // the index set is {10,15,20}
p[10] = 1;
p[15] = 5;
p[20] = 2;
p[30] = 3;             // sets p[20] to 3
x = p[10] + p[20];     // x is 4
```

The example introduced at the beginning of this section would now look like this:

```
array b[10];      // within a background program
                  // b[1], b[2], etc are entered by the subject
a = find ( type == A, a );
Profit = abs( a - b[a] );
```

### 3.13 Data use across periods and treatments

### 3.13.1 Definition of new tables

In z-Tree it is possible to define more tables than the standard subjects, globals, summary, contracts, and session tables. You can use these table for example for the following purposes:

– Multiple contracts-like tables for different markets.

– Random period payment: A table that contains the profits made in each period. This table can cover more than one treatment.

– Customizable history table.

– Storing a discrete function.

– Storing data of a payoff calculator.

– Present different decisions in a random order.

To use a new table, you have to define it at the beginning of the Background with the menu item Treatment → New Table…. A dialog opens where you can enter the properties of the table.
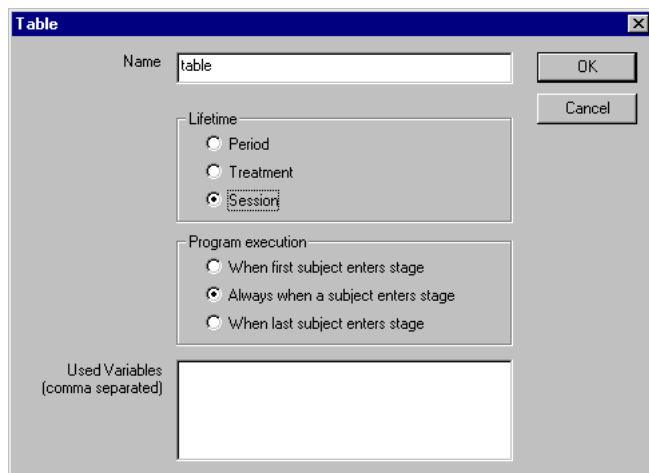
Figure 3.61. Table dialog

### Lifetime

If a table's lifetime is  Period , the table is reset whenever the end of a period has been reached, i.e. all records of the table are deleted (of course, after they have been saved), and the table is reconstructed from scratch. The subjects, globals, and contracts tables have a period lifetime. As in the subjects, globals, and contracts tables, the data that was in a table x in the previous period can be accessed in the table called OLDx.

Tables with lifetime  Treatment  are reset after the end of the treatment. They do not change after the end of a period.

Tables with lifetime  Session  are never reset. Nevertheless, they are stored at the end of each treatment. To be able to access variables defined in a treatment that was executed before, you have to declare the variables in the field  Used Variables .

### Program execution

If subjects do not enter a stage simultaneously, programs in a table can be executed whenever a subject enters, or only when the first enters the stage, or only when the last subject enters the stage. You can define the option for your databases. The programs in the globals table are executed for the first subject, programs in the summary table for the last subject. Programs in the subjects and in the session table are executed for each subject who enters the stage.

### 3.13.2 Accessing data from previous periods and treatments

If you want to program an auction market, you may want the subjects to keep their endowments of money and stock from period to period. In z-Tree, the subjects table is set up freshly in every period, so the information about earlier periods is not available directly. However, the tables of the previous period can be accessed with the prefix 'OLD'. So if you want to copy a variable Stock from the previous period, you write

```
if (Period > 1 ) {
    Stock = OLDsubjects.find( same( Subject ) , Stock );
}
```

There is no way of accessing data from even earlier periods.

If you need to access more information, you can use table with lifetime treatment or even session.

### 3.13.3 Copying data from treatment to treatment with the session table

Consider the following experiment: First you run a public goods treatment. The income earned in this public goods treatment can then be used in a second treatment in a market game. Because the session table survives the end of a treatment, you can solve the above problem elegantly. In the first treatment, you store the total profit in the variable Income in the session table and in the second treatment you can access this variable again.

How you can store data in the session table: In programs in the session table, the corresponding record of the subjects table can be accessed with the scope operator. Therefore, in your first treatment, you can enter the following line into a program of the session table:

```
Income = :TotalProfit;
```

How you retrieve data from the session table: You can access variables in the session table as in any other table. There is only one problem: z-Tree always checks that you do not use variables that are not defined earlier in the treatment. Therefore, you have to declare what variables in the session table you will use in a treatment. The clean way is to define the variables at the beginning of the treatment within the session table definition. All tables are listed at the beginning in the Background of the treatment. To define the variables you need to use in a treatment, you double click the session table icon and enter the variables into the field Used variables. A hack solution consists of defining the variable using an assignment statement. In order not to overwrite the data, you do the assignment in a statement that is not executed:

```
if (FALSE) {
    // assign
    VariableYouWantToUse = 0;
}
```

You can also use user defined tables to store the data and reuse it in another treatment.

If you use variables in the session table, it is usually optimal to do it with a program in the session table. In this case you can access the corresponding record in the subjects table using the scope operator.

## 3.14 Response time

Response time can be recorded on the server. This method is easy but because there are delay on the network and within z-Tree, more precise timing can be achieved using response time measurement on the client, i.e. on z-Leaf.

### 3.14.1 Time on server (z-Tree)

In order to measure time on the server, you use the function gettime(). It returns a the time in seconds that has been passed since z-Tree has been started. The resolution is milliseconds but for technical reason the precision is more around 10 ms.

Usually, you define a `StartTime = gettime();` e.g., when the stage start and determine the response time with `RT = gettime() - StartTime;`.

### 3.14.2 Time on client (z-Leaf)

The only way to get precise time - in particular time not distorted by network and server delays - is to store time on z-Leaf. There are three element that help you doing this.

1.  In all active elements, i.e., the buttons, plot inputs and input items, you can specify an event time. It this field, you enter the name of a variable. If the corresponding element is activated, e.g., if a button is pressed, the time (in seconds) since the start of the period is entered into this variable.

2.  If you put an ON/OFF trigger into a box, you can also specify an event time. This gives you the appropriate start time.

3.  In the background, you can specify a variable in which the start time of the period is stored. This is the absolute time, with date and time zone. Thus, this variable has to be a string variable.

Below you see a minimal working example for this method.
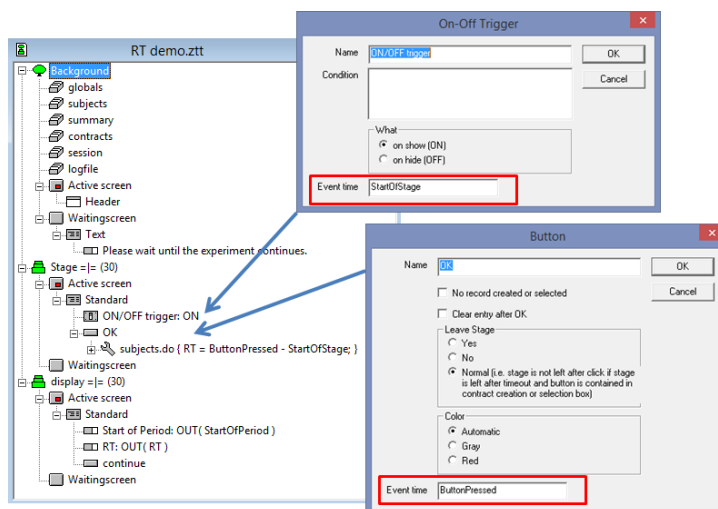


Figure 3.62. Demo treatment for measuring response time.

You can use the display condition to turn boxes on and off - and in this way to trigger the corresponding ON/OFF trigger.

## 3.15 Connecting to other software (and hardware)

There are many applications for connecting external hard and software. In particular, the use of psycho-physiological measurements as heart rate or eye tracking requires to synchronize the activities in z-Tree with the external stimuli and measurements. You can also use to delegate complicated calculations to an external software.

### 3.15.1 Linking time stamps

Very often it is sufficient to synchronize the activities ex post. In this case, you only have to know the absolute time of all events - both in z-Tree as well as in the external software. In z-Tree you can use the methods presented in Section Section 3.14.2, "Time on client (z-Leaf)". It is recommended to generate a file that contains all the relevant data in a format that is compatible with the external software. Often it is easier to generate this file already in z-Tree.

### 3.15.2 The connector

The connector provides a flexible and general method to link external software with z-Leaf. You can establish a socket connection between a z-Leaf and external software. This allows exchanging data and triggering events. There are three new item in the stage tree that provide the connections. The connector defines the connection, i.e., IP address and port of the target. The inlet allows to read data in from the external software. You specify the table and the variables the date is read in. New records are created for data that comes in. You can put a program into an inlet. This program is triggered when input arrives. The outlet allows to send data to the external software. You specify in item which data is sent to the external program.

The exchanged data is text based. It is composed of lines, which contain tab separated information. Each line starts with the keyword `z-Leaf Connector`, followed by a message ID. Then, the variables follow, also separated by tab.

We guide you now through an example, in which the external software is programmed in Python. It requires some basic programming knowledge of Python.

In our treatment, we can click on a point. This point is displayed and sent to the Python program. The Python program reads the point and generates P points around this point. These points are sent back to z-Leaf, which triggers a program that creates Z points around these points. We can change P and Z. P will be transferred to the Python program.

In our treatment, we define the following tables:

– parameters: contains the changes in P and Z. They have to be contained in a table to be transferred to the Python program. This table is also used to initialize the parameters on the Python side.

– clicks: contains the points that are clicked, with coordinates x and y.

– points: the points that are created by Python and the z-tree program, with coordinates x and y.

We create one stage. In the active screen, we put a standard box with input items for P and Z and four buttons. OUT leaves the stage and the treatment. Update changes the parameters P and Z. As a record is added to the parameters table, this information is transmitted to Python. Toggle Connect connects and disconnects. It changes the "condition" in the connector.
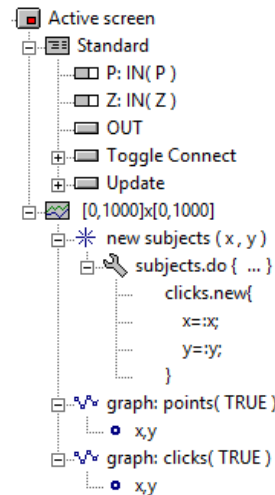


Figure 3.63. Active screen of the treatment with a box with buttons and a plot box.

These are the programs in these buttons:

```
// updateparameters.new{
P = :P;
Z=:Z;
}
// Toggle connect
if (connect == 0) {
  connect = 1;
  parameters.new{
    P = :P;
    Z=:Z;
  }
}
else {
  connect = 0;
}
```

The following figure shows the connector components in the stage tree.
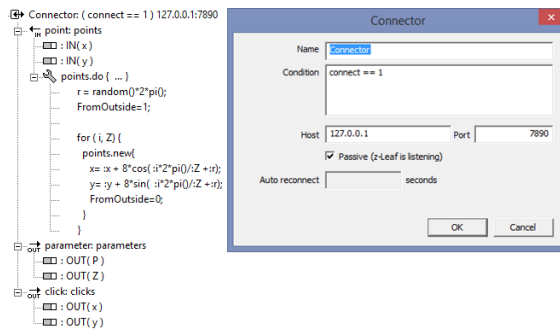
Figure 3.64. The connector components in the stage tree.

The connector tries to establish a connection to the specified IP port. In the example, port 7890 on the local computer (IP 127.0.0.1). If you want to connect to the local port, you can also leave the port open. It tries to establish a connection at the beginning of the stage in which this connector is placed and whenever the condition *changes* to TRUE. You can specify whether a connection is reestablished if the connection breaks downs. In "Auto connect", you can specify how often this is tried. You can also specify a listening connection. In such a connection the partner has to connect. The Outlets become active whenever in the respective table an entry is made. The first Outlet sends the parameters P and Z to Python (to port 7890 on IP 127.0.0.1) whenever a new entry is made in the parameters table. The second Outlet sends the parameters x and x to Python whenever a new entry is made in the clicks table. The Inlet deals with messages from Python. A "point" message with values x an z in entered into the points table. In addition, the program in the Inlet is called, which generates further Z points.

Below, you find the Python code that responds to z-Tree. It first listens for a program to connect. Then it waits for messages parses them and sends messages back.

```python
import socket
import time
import re
import math


P = 1;
Z = 1;


s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(('127.0.0.1', 7890))
s.listen(5)
print ('Listening on ', '127.0.0.1', ':', 7890)
```

```python
while True:
    c, addr = s.accept()
    print ('Connected by ', addr)

    while True:
        try:
            data = c.recv(4096)
        except (socket.error, e):
            print (e)
            print ('Connection closed')
            c.close()
            continue

        datastring= data.decode('UTF-8')
        lines = datastring.split('\n')
        print ( len(lines))
        numMsgs = len(lines)-1
```

```python
        for i in range(0, numMsgs):
          msg = lines[i].split('\t')
          if msg[0] != 'z-Leaf Connector':
           print ('Protocol error: Initialization message does not start with \"z-Leaf
 Connector\"')
          else:
            if len(msg) == 4 and msg[1] == 'parameter':
                P = int(float(msg[2]))
                Z = int(float(msg[3]))
                print ('Initialization message received; P = %d; Z= %d' % (P, Z) )
```

```python
            elif len(msg) == 4 and msg[1] == 'click':
                x1 = int(float(msg[2]))
                y1 = int(float(msg[3]))
                print ('Points message received; x = %d, y = %d' % (x1, y1) )
                for j in range(P):
                    try:
                        dx = 25*math.cos(j*2*math.pi/P )
                        dy = 25*math.sin(j*2*math.pi/P )
                        outbytes = ('z-Leaf Connector\tpoint\t%d\t%d\n' % (x1 +dx,
y1+dy)).encode('UTF-8')
                        print ('sending %s' % (outbytes) )
                        c.send(outbytes)
                    except (socket.error, e):
                        print (e)
                        print ('Connection closed')
                        break
                    print( j)
```

```python
        else:
          print ('Protocol error: Expected an message \"init\" with two values')


            print ('Using default values')


        time.sleep(0.1)
    c.close()
    [source,python]
```

# 4 Questionnaires

## 4.1 Overview

Each session ends with a questionnaire. Generally, name and address are asked first so that the *payment file* can be written. This is the file that contains a list of the names and earnings in the experiment. It is not necessary to include names if you have privacy concerns, but an option. Next, additional question-naires are inserted. Then a screen that displays the subject's earnings is inserted and finally a good-bye screen appears where subjects get instructions where to get their payment.

A questionnaire consists of a series of *question forms*. One question form displays the individual *questions* in the same way as the items in the standard box of a treatment. However, the questions do not appear vertically centered and adjusted to the screen size. Therefore, question forms may be larger than the screen. If this is the case, a scrollbar appears. The answers in questionnaires are of no consequence, which means that they are not entered into a database. The answers are only saved as text and cannot be used in programs.

The question forms are worked through one after the other. When all subjects have answered all question forms they are in the state 'ready' and you can start a further treatment or a further questionnaire. It is generally impractical to start several questionnaires one after the other because this leads to unnecessary waits. It is better to integrate all forms into one questionnaire file.

The last question form remains on the users' screens until you continue, i.e., until you start a new treatment or questionnaire or until you shut down the computer. Therefore, the final question form must not contain a button.

## 4.2 Making questionnaires

### 4.2.1 Address form

The address prompt asks for the subject's address. The fields First Name, Last Name and Continue (button label) are compulsory if the address form should appear. The other fields are optional. If the question texts in the optional fields are left empty, the corresponding question does not appear on the clients' screens. If the fields First Name and Last Name are left empty, then no form appears. The payment file is written whenever the last subject has completed (or passed through without completing) the address form. Additional variables from the `session` table are added to the address form if questions are placed into the address form.

When all subjects have passed through the address form, the payment file is written. This is a tab-sep-arated file containing computer identification, earnings from the experiment, and possibly names. This file can be printed and payment of the subjects can be based on this list. In Section 4.3, "Running a questionnaire", we describe how to produce individual receipts.

### 4.2.2 Question forms

Question forms consist of a list of questions. The layout of the questions is adjusted with rulers. All question forms except the last must contain a button.

If you have conducted an experiment where subjects played different roles, for instance, where there were proposers and responders, then you may also want to use different questionnaires for motivational questions. You can *omit* question forms. If you want to present two different questionnaires to two different types of subjects, you put the question forms for both types into one questionnaire document and omit the forms selectively for the type you want.

The procedure to omit question forms is similar to the procedure to omit stages in a treatment: You set the variable `Participate` to zero if you do not want the subject to fill out that question form. Since there is no subjects table available when you run a questionnaire, the variable `Participate` has to be set in the session table. You can do this in the program that can be entered in the question form dialog.

> In the treatment, you have to copy the type variable (the variable that determines whether the subject is a proposer or a responder) into the session table.

### 4.2.3 Questions

Questions in questionnaires correspond to items in treatments. In items, display options are defined in the field layout. Here they can be set with radio buttons. The fields are as follows:

Label

> Name of the question as it is shown to the subjects. If no variable is set, the label appears as text over the whole width of the screen.

Variable

> If it is an input variable, this is the name of the question as it should appear in the data file. If input is not set, it has to be a variable of the `session` table.

Type

> Layout of the question.
>
> Text, Number
>
> > A text field appears into which input is entered. In the case of the number option there is a check to see if it is really a number that has been entered. Furthermore, the check determines whether the number is in the valid range. If the option Wide is selected for a text field, the entry field may consist of several lines. The number of lines can be selected.
>
> Buttons
>
> > A button is created for every line in the field Options. Of course, only one question with this option is permitted per question form.
>
> Radiobuttons
>
> > A radio button is created for every line in the field options. These buttons are arranged vertically one on top of the other.

Slider, Scrollbar

The slider and scrollbar make a quasicontinuous entry possible. Minimum, maximum, and resolution need to be entered. In the wide layout, a label can be placed at the margins. These labels are entered in the field options. The maximum is always the value at the right border, the minimum is the value at the left border. Maximum may therefore be smaller than minimum.

Radioline

A radio button appears for all possible answers determined by minimum, maximum and resolution. In the wide layout, a label can be placed at the margins. These labels are entered in the field Options. The middle button can be separated from the others by entering a number greater than zero for Distance of central button. The maximum is always the value at the right most button, the minimum is the value at the left most button. Maximum may therefore be smaller than minimum.

Radiolinelabel

Label line for radio button lines. Labels can be placed above the buttons at the margins. The texts for this action are in the Options field.

Checkbox

A checkbox is created for every line in the Options field. These checkboxes are arranged vertically one on top of the other. The resulting value is a list of all options checked.

Wide

In the wide layout, the entry region for the question appears under the label over the whole width and does not only appear on the right of the label in a second column.

Input

This field determines whether a question is presented to the subjects (input is set) or whether a variable is only shown (input is not set).

Empty allowed

This is set when the question does not need to be answered.

Minimum, Maximum, Resolution

When numbers have to be entered, these values are used for checking. With sliders, scrollbars and radio lines these values are used for converting the position into a number.

Num. rows

In the wide layout for text entries, this is the number of rows that can be entered.

Options

The labels of buttons, radio buttons and, in the wide layout, of sliders, scrollbars and radio lines.

Figure 4.1. Layout of questions, normal layout (left) and wide layout (right)

### 4.2.4 Profit display

If you wish to display the earnings made in the session to the subject, you use a question form where you can also show variables from the session table. These include:

`FinalProfit`

Sum of all earnings from the treatments.

`MoneyAdded`

Money injected by the subject (see bankruptcy procedure).

`ShowUpFee`

Amount of show-up fee.

`MoneyToPay`

`Showupfee` + `FinalProfit` + `MoneyAdded`.

`MoneyEarned`

`Showupfee` + `FinalProfit`. This is the value used in the payment file.

### 4.2.5 Rulers

Rulers are used to set the regions where labels and questions are positioned.

Distance to left margin

The position can be given in points or in percentage of the screen width.

Distance to right margin

The position can be given in points or in percentage of the screen width.

Distance between label and item

This is the horizontal distance between the label and the question. Half of this distance is also kept to the left and right margins.

Size of label

This is the maximum width of the label. When the display is given in a percentage, this refers to the width between the left and right margin and not to the width of the whole screen.

### 4.2.6 Buttons

Buttons conclude question forms. You cannot define more than one button per form. However, this button can be positioned anywhere on the form. The label of a button can be selected by you.

### 4.3 Running a questionnaire

At the end of a session, the experimenter always starts a questionnaire with the command Run → Start Questionnaire which is described below. This questionnaire must include an address form (a prompt for the subject's address, or a 'virtual' step if you do not wish to ask subjects for their address information) in order to write the payment file. When all subjects have gone through the address form, z-Tree writes the payment file. You may also insert a questionnaire at another point in the session. However, as questionnaires do not contain information regarding the number of subjects, they can be started, at the earliest, after the first treatment (e.g., the welcome treatment). If more than one questionnaire contains an address form, the address only needs to be entered in the first questionnaire. In subsequent questionnaires only the payment file is updated.

A questionnaire is started with the command Run → Start Questionnaire. This command is located at the place where the command Run → Start Treatment is located when a treatment window is open. While subjects are answering the questionnaires, the state in the *Clients' Table* shows which questionnaire is currently being answered.

### 4.4 Making individual receipts

The payment file is a good basis to produce individual receipts using a mail merge program, which is contained in most word processors. It is a good idea to produce a step by step procedure for your environment that is somewhere available in the lab.

# 5 Conducting a Session

## 5.1 Quick guide

The running of a session consists of the following steps:

1.  preparation of treatments and questionnaires

2.  startup of the experimenter PC

3.  startup of the subject PCs

4.  arrival of subjects

5.  start of the session and the first treatment

6.  observing the course of the session

7.  start further treatments

8.  conclusion of session with a questionnaire

9.  payment

10. switching off of machines

11. data analysis

While conducting an experiment, we mainly work with the menu Run. The order of commands in the run menu corresponds, on the whole, with the course of the session.

## 5.2 Preparation of treatments and questionnaires

Before the session begins, all treatments and questionnaires in z-Tree are defined and saved on the file server. How you define treatments and questionnaires is explained in Chapter 3, *Definition of Treatments* and Chapter 4, *Questionnaires*. Treatments are different for different numbers of subjects. Hence, in case you are not sure whether all subjects will show up, you would like to have the possibility of running a session with varying numbers of subjects - depending on how many show up. In simple cases the number of subjects, which is fixed in every treatment, can be changed at the beginning of the session, i.e., before the start of the first treatment. In cases where changing the number of subjects involves more than simply changing this number, you have the option of preparing treatments for different numbers of subjects.

We recommend that you make a list of the treatments and questionnaires. It should include the names of all of the treatment files for all possible numbers of subjects. Alternatively, you may list how the treatment files are to be adjusted for all possible numbers of subjects. It is a good practice to number the treatments, e.g., `01_welcome.ztt`, `02_PG.ztt`, `03_SVO.ztt`, and `04_questionnaire.ztq`.

### 5.3 Start-up of the experimenter PC

Switch on the experimenter PC, log in and start z-Tree.[1] If you want to start z-Tree with a special option, for example with respect to where to store the data, use a command file to start z-Tree. z-Tree has a window that displays which clients have started up and established contact. This information is displayed in what is called the *Clients' Table*, which will be described in more detail in the next section.

### 5.4 The clients' table

The clients' table shows which clients are connected to the server. It also contains information about the state of the clients, i.e., which screens are currently being displayed to the clients. You may open the clients' table with the first command from the Run Menu.

Each line corresponds to a subject. This subject is seated at a PC on which the client is running (column 'Clients'). As long as no treatment is started, the clients can be moved to other lines. They can be sorted and shuffled with the commands Run → Sort Clients and Run → Shuffle Clients respectively. As soon as you start a treatment, the order of the clients is fixed.

The clients' table has three columns:

client

> Name of the client. In general this is the name of the subject PC. If a client is no longer connected, its name appears in brackets. If a client with the same name should reconnect, it can continue at the same place where the previous subject left off. The caption to the clients' table gives the number of clients currently connected.

state

> The state in which the subject is at the time. Example: 'Ready' when the subject is waiting for the next treatment or for the next questionnaire.

time

> Displays the remaining time as it is shown to the subjects.

### 5.5 How a client establishes a connection with the server

A session begins when the experimenter starts the server program z-Tree on the experimenter PC. After this, the client program z-Leaf is started up on the subject PCs. z-Leaf establishes contact with z-Tree on the experimenter PC. The communication between z-Tree and z-Leaf is based on TCP/IP.

### 5.5.1 How does the client know the server's address?

z-Leaf can determine the server's IP address with the command line option `/server ipaddress`. If, for instance, z-Leaf is started with the command

```
zleaf.exe /server 100.20.10.233
```

[1]The name under which one logs in depends on the installation. Typically one will set up a user account for all persons who carry out experiments.

then z-Tree must run on a PC with the IP address 100.20.10.233. This method is very useful if you want to run an experiment without a file server. If z-Tree is located on a file server, however, there is a more comfortable way: z-Tree automatically writes it's IP address into the current directory in the file `server.eec`. If z-Leaf is started from the same directory (on the same PC or on another) it reads this file. If z-Leaf does not find this file, it looks for it in the directory `c:/expecon/conf` on its computer. If it is unable to find either, it can take the IP address of the local machine. It can now seek the server at this address. If the server is not found here either, the person who starts the client is asked if he or she would like to try again.

### 5.5.2 Channel and TCP ports

If you want to run more than one z-Tree on one computer they must work with different channels, i.e., you set the channel with the command line option `/channel ch` to values greater than zero.

The channel determines the TCP port used in z-Tree to listen for connection attempts from the z-Leaves. The actual port is 7000 + channel (defaults to channel 0, port 7000).

z-Leaf listens on port 7999 or lower for a restart message.

### 5.6 How to set the name of a client

If z-Leaf is started up by means of a name on the command line (e.g., `zleaf.exe /name T1`), this will be the name of the client (e.g., T1). Otherwise the name will first be sought in the file `name.eec` in the local directory. This way you can run several clients on one PC. If the file `name.eec` does not exist, the host name of the PC is used. The host name is entered in the network control panel under TCP/IP.

### 5.7 During a session

In the windows opened with Run → globals Table, Run → subjects Table, Run → contracts Table, and Run → summary Table the content of the given table is displayed. In these tables the sizes of lines and columns can be changed. These tables are automatically exported to the *xls file*.

### 5.8 Concluding a session

Each session ends with a *questionnaire*. Generally, the address is asked first so that the *payment file* can be written. Next, additional questionnaires are inserted and finally a good-bye screen appears which also displays the profit. How questionnaires are made is explained in chapter Chapter 4, *Questionnaires*.

### 5.9 Dealing with a crashed or blocked subject PC

All data is stored in various files in readable form as soon as it is complete. Besides this, all messages exchanged between server and clients are stored in a (non-text) file, the *GameSafe*. With the help of these files it is generally possible to continue the experiment after a computer crashes.

In addition, the GameSafe can be used to recover variables that are overridden. The GameSafe is stored in binary format. It can be transformed into a readable text format by "exporting" it. Please use the File → Export → GameSafe.

In the case of a problem on a subject PC, you proceed as follows: If the PC is still working, it can simply be restarted. It is then automatically reset to the same state it would have been in, had the crash not occurred. Of course, this also means that it can be further than before: If, in the meantime, the time set for a stage has passed, then the treatment moves ahead by a stage, even if not all the clients are connected.

There are several possible reasons why a subject PC may not resume a treatment. It is possible that the server is very busy. In this case you simply have to wait a little. However, if the server is in a wrong state due to a programming error, the server has to be restarted.

If the subject PC is not functioning, you can start a new PC. As soon as the new client appears, it can be manually assigned to the subject who was working on the old computer. To do so, you select the client field of the new client in the clients' table and move it over the field of the old client.

## 5.10 What to do if the experimenter PC crashes

After a crash of z-Tree, you have to do the following:

1. Restart z-Tree.

2. Open the clients' table.

3. Restart all clients with the Run → Restart All Clients. If some clients do not connect, go to the client and reconnect the client manually. To reconnect a client manually, you quit z-Leaf with **Alt+F4** and start z-Leaf again. If no client connects, then you have four possibilities:

   – You can restart all clients manually.

   – You can wait a while and try to restart the clients later. (It can take up to 4 minutes.)

   – You can shut down and restart the experimenter PC. (Exiting windows and logging on again is not sufficient.)

   – You can start z-Tree on another computer.

4. With Run → Restore Client Order you sort the clients in the same order as they had been in the crashed session.

5. With Run → Reload database you restore all tables. Since the tables are stored after each period you can restore the state when the last period was finished.

6. Check how may periods have been played. You find this information for instance in the summary table or in the subjects table.

7. Open the treatment you were running before. If n periods have been played, set the number of practice periods to –n (minus n).

8. With Run → Start Treatment you start the treatment again.

## 5.11 What happens if subjects suffer losses?

Losses by subjects can be covered by the following sources:

– Previous profits.

– Lump-sum payment.

– Show-up fee.

– Money injected during the session.

When a subject suffers losses, messages appear on the screen. The text of these messages is determined in the treatment (in the Background).

If the first two sources cannot cover the losses, but the show-up fee can, a message appears on the subject's screen. It informs the subject that he or she can now choose either to use the show-up fee or drop out of the game. If the subject uses the show-up fee, he or she may simply play on.

If, on the other hand, the subject chooses not to use the show-up fee to cover his/her losses, he or she reaches the state 'BancruptShowupNo' and you have to release the subject from the server. If the show-up fee is used and exhausted, a message appears on the subject's screen informing him/her that he or she has incurred a loss. This message can be concluded with a question. If the subject answers this question with "yes", he or she arrives at the state 'BancruptMoreYes'. By answering "no", he or she arrives at the state 'BancruptMoreNo'. In this case also, the experimenter has to release the subject from the server. A dialog appears by double-clicking the 'State' field. There are three possibilities:

– You allow the subject to continue. In this case you need to type a number into the field Amount injected that is higher than the current loss. You either make the subject pay this amount or, alternatively, consider it to be the credit limit. You take it into account when ascertaining whether subjects' earnings are negative. However, it is not shown together with the subject's profit, nor is it considered in the payment file. This means that you need to add the amounts injected to the amount indicated in the payment file. In this way, the payment file lists the net amounts paid out to the subjects.

– Another subject takes on the role of the subject released. In this case, all profits are reset to zero and the new subject is able to continue the experiment at the PC of the released subject.

– The subject drops out. You need to be careful about this as it may change group sizes and the information on the active subjects may therefore no longer be correct.

## 5.12 The files generated by the server

The following files are created in the course of the experiment. They are distinguished by their suffixes. All files are *tab separated* text file. The use of the extension 'xls' only allows to open the file directly in Excel. Do not open and save your files under the original name. Sometimes Excel creates dates out of numbers like 1.2. It is easier to restore the data from the original file. In order to avoid the 'date' problem you have to instruct Excel to use the point for decimal numbers and not the comma.

`.pay`

Payment file. Name of client computer, profit, and possibly subject name.

`.adr`

    Addresses of the subjects.

`.sbj`

    Questionnaire responses. Without subjects' names.

`.xls`

    Contains all tables shown in the course of the session. The first column contains the treatment number, the second the name of the table and the third column contains the period variable. As the tables are stored chronologically, you need some preparation to work with them. A starter on how to prepare the data is given in Section 5.13, "Data analysis".

`.gsf`

    *GameSafe* in binary form. It can be exported into a readable file but this file will be gigantic.

The starting time of the session is used as file name. The format is YYMMDD_HHMM. YY means year, MM month, DD day, HH hour and MM minutes. In this way the start of the session can easily be reproduced.

### 5.13 Data analysis

The xls file can easily be used for data analysis, and there are tools in the Tools → ] menu that facilitate this process considerably. These commands allow you to separate the different tables into separate files and to merge similar files into a big file for analysis. The best way to do this is to copy all the data files into a new directory → then apply menu:Tools[Separate Tables… to all files. You can select all files in one dialog by holding the **Ctrl** key when clicking. Then you apply the Tools → Join Files… command. A dialog appears and in it you add all files which have a similar structure – for instance all subjects tables. It is not necessary that all merged tables have the same variables. The merge tool creates a new table that contains all variables that appear in any of the tables.

# 6 Installation and lab architecture

## 6.1 A simple installation with a file server

The easiest way to install z-Tree is to put z-Tree and z-Leaf in a common directory on a file server. The server must have both read and write access to the directory while the client should only have read access. When z-Tree starts, it writes the IP address of the server PC into the file `server.eec` in this directory. The z-Leaves read the address of the server in this file and connect with z-Tree on this machine.

## 6.2 Installation without a file server

z-Tree and z-Leaf can run on different computers that do not have access to a common file server. This option is needed if you want to run experiments over the Internet. As opposed to an installation with a file server there is no easy method for z-Leaf to know where z-Tree is running. Therefore, z-Leaf must have the file `server.eec` (the file contains the IP address of the server) in its directory or in the directory `c:/expecon/conf`. z-Tree's IP address can also be set with the command line option `/server ip-address` when the client starts up.

If z-Tree and z-Leaf do not run from a common directory, the function Run → Restart All Clients does not work. If there is a problem, the clients have to be started up manually.

## 6.3 Setting up a test environment

You do not need several computers to test programs. You can start z-Tree and more than one z-Leaf on one computer. However, you have to give the z-Leaves different names. For this you use the command line option `/name`. For example, you can start z-Leaf twice, once with `zleaf.exe /name A` and once with `zleaf.exe /name B`.

## 6.4 Running an experiment in different languages

All commands in z-Tree are in English. Most texts for the subjects are defined by the experimenter and can therefore appear in any language. The error messages in z-Leaf, however cannot be overridden. The default language in z-Tree is German (the language written in Zurich). You can change the language in z-Tree in the menu or with command line options. To change the texts in z-Leaf, you have to use the command line option. It has the form `/language lan`. `lan` can have different values such as 'en' for English or 'de' for German (see all available languages in the reference manual). If you want to change the language in your environment, you can create a shortcut for z-Tree that initializes the language to your choice.

## 6.5 Running more than one z-Tree

If you want to run more than one z-Tree on one computer, the different instances must use some identification to be distinguished from each other. This distinction is called the *channel*. The channel is set with the command line option `/channel ch`. The z-Leaves and the z-Tree that belong to one session must use the same channel.

## 6.6 A sample installation

In this section, we describe how z-Tree is installed in the lab at Zurich. Figure 6.1, "Lab setup in Zurich" shows the directory structure of the installation in Zurich.
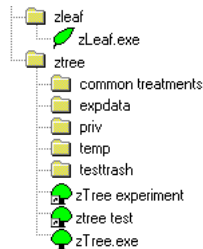


Figure 6.1. Lab setup in Zurich

We use the command line options to tell z-Tree where the files should be stored. These options are:

| | |
|---|---|
| `/xlsdir` | Sets the directory where the xls file is stored. |
| `/sbjdir` | Sets the directory where the subjects file is stored. |
| `/datadir` | Sets the directory where the xls and the subjectsfile are stored. |
| `/adrdir` | Sets the directory where the address file is stored. |
| `/paydir` | Sets the directory where the payment file is stored. |
| `/privdir` | Sets the directory where the address and the payment file are stored. |
| `/gsfdir` | Sets the directory where the GameSafe is stored. |
| `/tempdir` | Sets the directory where the temporary files `@lastclt.txt`, `@db.txt`, and `@prevdb.txt` are stored. |
| `/datadir` | Sets the directory where the treatment and questionnaire files are automatically stored (before they run). |
| `/leafdir` | Sets the directory where the file server.eec is stored. |
| `/dir` | Specifies a directory where z-Tree put all files. |

The shortcut 'zTree experiment' starts z-Tree with the following command line options:

```
/datadir expdata /leafdir ..\zleaf /privdir priv /tempdir temp /gsfdir temp
```

Shortcut 'ztree test' is used for testing and starts z-Tree with:

```
zTree.exe /dir testtrash /leafdir ..\zleaf
```

Experimenters have read and write permission in all directories. Subjects have read permission in the `zleaf` directory.

We have placed z-Leaf in a different directory from the directory where z-Tree is located. The subjects only have access to this directory and they have only read access to it. This prevents subjects from getting access to your experimental data. To be able to automatically start the z-Leaves wherever z-Tree is started, we set the leaf directory in the options of z-Tree.

# Reference Manual

# 7 The Stage Tree Elements

This chapter explains options of the stage tree elements that cannot be created. The stage tree elements that can be created are explained in the respective menu section.

### 7.1 Editing in the stage tree

### 7.1.1 Viewing

Branches of the stage tree can be opened and closed.

### 7.1.2 Adding elements

Stage tree elements are added from the menu.

### 7.1.3 Deleting elements

Stage tree elements can be deleted using the Cut-command **Ctrl+X**, but not with the delete key **Delete**.

### 7.1.4 Moving elements

Elements and branches can be moved with the mouse. When it is ambiguous whether an element is moved after or within an element then moving the element onto the *icon* moves it *after* the element; if it is moved onto the *text*, it is moved *into* the element.

### 7.1.5 Editing elements

The parameters of stage tree elements are edited in the dialog. The dialog can be opened using the Treatment → Info… menu command or with a double click.

### 7.2 The background

Information not specific for a particular stage may be inserted and defined in the Background. First, the background contains a list of tables that are used for calculations. Then, in the screens of the background, elements can be placed that are used in all stages. In our example, all active screens have a header window and all waiting screens a text "Please wait until the game continues". The programs of the background are run at the beginning of a period. They are used for defining constants. In the background itself, i.e., when you double-click it, some central parameters of the treatments can be viewed and changed:
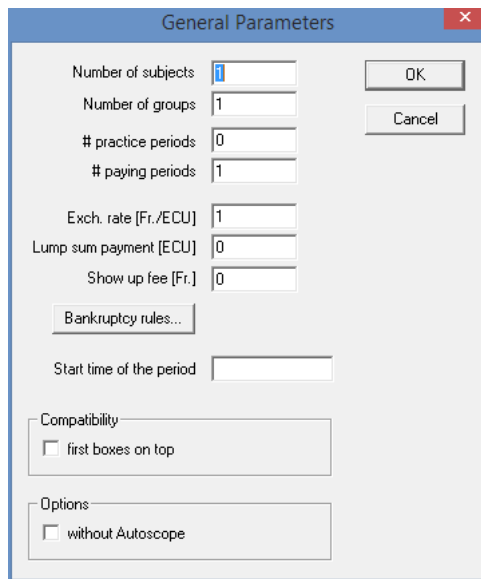
Figure 7.1. General parameters dialog

## Number of subjects

This number is constant for a whole session. Therefore all treatments must have the same number of subjects.

## Number of groups

Number $g$ of groups to which the subjects are assigned. This means that the subjects belong to a group with an ID between 1 and $g$. This value is used for the matching commands and for checking.

## # practice periods

Number of periods in which no profit is paid out. The number entered can be negative. In this case the period counter starts later. If, for instance, the number of trial periods is -2 and the number of paying periods is 10, then the treatment extends from period 3 to period 10.

## # paying periods

Number of periods in which the profit is paid out.

## Exch. rate [Fr./ECU]

How many Francs ($, €, £, …) are paid out per *experimental currency unit*.

## Lump sum payment [ECU]

Amount in internal units that is credited at the beginning of period 1 of the treatment. This amount figures in the variable `TotalProfit`.

## Show up fee [Fr.]

Amount credited to the subjects at the start of the session. Only the show-up fee of the first treatment played is decisive.

Bankruptcy rules…

By clicking this button the parameters of the bankruptcy rules may be adjusted, see Section Section 7.2.1, "Bankruptcy rules". In the tutorial, you get a detailed description about how to deal with losses.

Start time of the period

May contains a variable name. The corresponding variable is created in the subjects table and is filled with the absolute start time of the period measured on the client as a string using the ISO 8601 notation. Note that the variable will be filled not before the client entered the first stage.

Compatibility: first boxes on top

By default the boxes on a screen are sequentially placed on top of each other. Thus, the last boxes are on top. Checking this box reverses this behavior, which results in first boxes being on top. (This was the default behavior in earlier versions of z-Tree.)

Options: without autoscope

This option affects the processing of programs. Normally, autoscope is on, which means that if a variable does not exist in a table, the system tries to find it in the scope chain. Turning autoscope off requires a stricter setting of the scope, which can help detecting errors.

### 7.2.1 Bankruptcy rules

Figure 7.2, "Bancruptcy rules dialog" shows a screenshot of the bankruptcy rules dialog. The first question appears when the subject has suffered losses that can be covered by the show-up fee. The second message appears when the losses suffered by the subject exceed the show-up fee. The fields near "yes" and "no" contain the labeling of buttons. If both buttons are not empty, the user has a choice. If one of the two buttons is empty, the user can only confirm the given answer. Depending on what button the user has clicked, the action stated beside the button is executed.
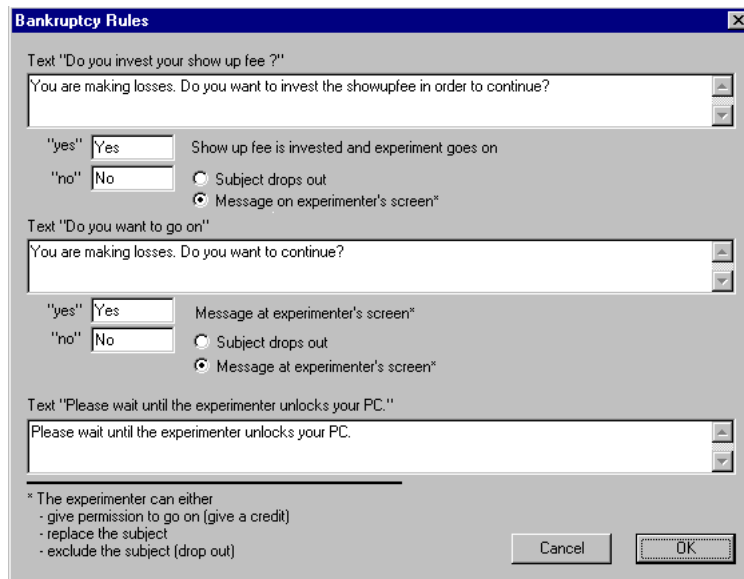
Figure 7.2. Bancruptcy rules dialog

### Show-up fee is invested and the game continues

The show-up fee is used to cover the losses and the game continues.

### The player drops out automatically (OUT)

By putting the subject into the state OUT, he or she is taken out of the game. This is not wholly un-problematic - the programs need to be equipped for this process. You can, for example, set an input variable to an impossible value in order to be able to discern that a subject has dropped out.

### Message to the experimenter

No message appears at the client's screen. The experimenter sees in the state of the subject that she is making losses.

## 7.3 Parameter Table

In the parameter table period and subject specific parameters can be specified in a separate table. In particular, group matching can be defined in this table. Group assignment is executed first. Then, the programs in the background are executed. Then, the programs in the paramter tables are executed; first the program in the specific parameters, then the programs in the role parameters and at last, the program in the period parameters. Finally, the programs in the first sage are executed.

We recommended to use the parameter table consistently, either always or never. Actually, never is probably the better choice. With the exception of perfect stranger matching, everything can be done in programs in the stage tree. Because you work in the stage tree anyhow, these programs are more transparent than definitions in the parameter table.

### 7.3.1 Period Parameters

Prompt

If a text is entered here, it is displayed on the server before the start of the period. The treatment will only continue when OK is clicked on the server.

Program

This program is run in the globals table after the programs in the background, the specific parameter program and the role program have been run.

### 7.3.2 Role Parameters

Name

Of no consequence.

Program

This program is run in the subjects table, after the programs in the background and the specific parameter program have been run.

### 7.3.3 Specific Parameters

Group

Group number of the subject in this period.

Name

Of no consequence.

Program

This program is run in the subjects table after the programs in the background have been run.

### 7.4 Screen

Background screen is being used

If this field is marked, the windows of the background are taken over and placed first. In every stage there are two screens. The active screen is shown to the subjects who participate in the stage until they have concluded the stage. The waiting screen is shown when subjects wait for the next stage.

### 7.5 What can be contained in stage tree elements?

In the following, we list for all stage tree elements what kind of other elements they can contain. We use the notation a < b c if the element a contains the element b followed by c. A star * after an element means zero or one or more instances of this kind of element. The first line therefore signifies that the

The Stage Tree Elements
What can be contained in stage tree elements?

z-Tree 4.1
Reference Manual

stage tree contains one background followed by some stages (zero or more). The notation `a = b | c` means that the term a is one of b or c.

```
stagetree           < background stage*
background          < table* programitem* connector* activescreen waitingscreen
stage               < programmable* connector* activescreen waitingscreen
programmable        < program
                    | externalprogram
                    | tableloader
                    | tabledumper
program             < -
externalprogram     < -
tableloader         < -
tabledumper         < -
connector           < connectoreinlet
                    | connectoreoutlet
connectorinlet      < item* programmable*
connectoroutlet     < item*
activescreen        < box*
waitingscreen       < box*
box                 = standardbox
                    | headerbox
                    | helpbox
                    | containerbox
                    | gridbox
                    | calculatorbuttonbox
                    | historybox
                    | contractcreationbox
                    | contractlistbox
                    | contractgridbox
                    | messagebox
                    | multimediabox
                    | slideshowbox
                    | chatbox
                    | plotbox
standardbox         < item* button*
headerbox           < -
helpbox             < -
containerbox        < box*
gridbox             < item* button*
calculatorbuttonbox < -
historybox          < item*
contractcreationbox < item* button*
contractlistbox     < item* button*
contractgridbox     < item* button*
messagebox          < -
multimediabox       < -
```

The Stage Tree Elements
What can be contained in stage tree elements?

z-Tree 4.1
Reference Manual

```
slideshowbox        < slideshowelement*
slideshowelement    = slide
                    | slidesequence
slidesequence       < slide*
slide               < plotbox*
chatbox             < checker* programmable*
plotbox             < plotitem*
plotitem            = plotpoint
                    | plotline
                    | plotpie
                    | plotrect
                    | plotaxis
                    | plottext
                    | plotgraph
                    | plotinput
item                < checker* programmable*
button              < checker* programmable*
plotpoint           < -
plotline            < -
plotpie             < -
plotrect            < -
plotaxis            < -
plottext            < -
plotgraph           < plotitem*
plotinput           < checker* program*
```

# 8 Menu Commands

## 8.1 File Menu

### 8.1.1 New Treatment

A new treatment is created. It contains the background and the background screens.

### 8.1.2 New Questionnaire

A new questionnaire is created. It is completely empty.

### 8.1.3 Open…

Opens a treatment or a questionnaire. A dialog in which the treatments are shown appears. In order to open a questionnaire, you have to select the questionnaire file type.

### 8.1.4 Close

Closes the frontmost window.

### 8.1.5 Save

Saves a treatment or a questionnaire.

### 8.1.6 Save As…

Saves a treatment or a questionnaire under a new name or at another place.

### 8.1.7 Export Treatment…

A treatment is written in textual form in a file. A treatment exported in this way may also be read in again.

### 8.1.8 Export Questionnaire…

A questionnaire is written in textual form in a file. A questionnaire exported in this way may also be read in again.

### 8.1.9 Export GameSafe…

A GameSafe file is translated from an unreadable binary format into a gigantic ASCII file.

### 8.1.10 Export Table…

A table is written in a tab-separated file.

### 8.1.11 Import…

Imports a treatment or a questionnaire.

### 8.1.12 Page Setup…

Not yet implemented.

### 8.1.13 Print…

Not yet implemented.

### 8.1.14 Previous files

Opens a treatment or questionnaire previously used.

### 8.1.15 Quit

Quits the program. A warning message appears if the session is still running, i.e., if a treatment is running, if a questionnaire is running or if the payment file is not yet written.

## 8.2 Edit Menu

### 8.2.1 Undo

Not yet implemented.

### 8.2.2 Cut

Deletes the marked branch in the stage tree or in the questionnaire and places it onto the clipboard.

### 8.2.3 Copy

Copies onto the clipboard the marked area of a table or the selected branch in the stage tree or in the questionnaire.

### 8.2.4 Paste

In the parameter table: Copies a table from the clipboard and pastes it at the selected position.

In the stage tree or in the questionnaire: Copies the branch from the clipboard and pastes it at the selected position.

### 8.2.5 Copy groups

This is a command for the parameter table. Copies the groups of the selected area of the parameter table onto the clipboard.

### 8.2.6 Paste groups

This is a command for the parameter table. Copies the groups from the clipboard and pastes them at the selected position in the parameter table.

### 8.2.7 Insert cells

This is a command for the parameter table. Adds more rows or columns immediately after the last row or column selected. If more than one row or column is selected, a corresponding number of rows or columns is added.

### 8.2.8 Remove cells

This is a command for the parameter table. Removes complete rows or columns. Complete rows are selected by clicking at the small cell at the left border of the parameter table. Complete columns are selected by clicking at the cell at the top of the parameter table.

### 8.2.9 Find…

Not yet implemented.

### 8.2.10 Find Next

Not yet implemented.

### 8.3 Treatment Menu

### 8.3.1 Info…

Opens a dialog in which the parameters of the selected element can be viewed and changed.

### 8.3.2 New Stage…

A stage corresponds to a screen that a subject has to go through. It actually consists of two screens, the active screen the subject goes through and the waiting screen that is shown when the subject leaves the stage and waits for the next stage. In the following, we show the stage options.
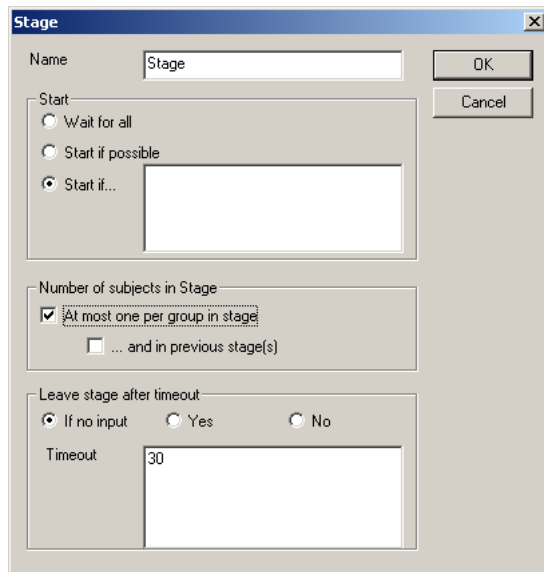
Figure 8.1. Stage dialog

### Name

Name of the stage. Used for documentation and as a name for the state in the clients' table.

### Start

#### Wait for all

Subject may not enter this stage if not all subjects are at least in the waiting state of the preceding stage.

#### Start if possible

No condition for subjects to enter this stage.

#### Start if

A condition can be given that must be satisfied before subjects can enter the stage. This condition is calculated in the subjects table. Be careful that this condition is eventually satisfied. Otherwise the experiment will never continue.

### Number of Subjects in Stage

#### At most one per group in stage

If checked only one subject per group can be in this stage. This option is used in posted offer markets.

#### … and in previous stage(s)

If the checkbox is checked, this stage and the preceding stage form a set. If this box is checked in the previous stage too, then the stage before also belongs to this set and so forth. Now, only one subject per group may be in any stage of such a set.

Leave Stage after timeout

If no input

The stage is terminated after timeout if no input is expected.

Yes

The stage is terminated after timeout even if input is expected.

No

The stage is not terminated after timeout.

Timeout

The number of seconds after which the timeout occurs. This expression is calculated in the globals table.

In the stage tree, the stage options are displayed after the name. The following notation is used:

| Option | Notation |
|---|---|
| Wait for all | $=\|$ |
| Start if possible | – |
| Start if | `(condition)` |
| Not at most one per group in stage | = |
| At most one per group in stage | – |
| … and in previous stage(s) | –…– |
| If no input, leave stage after timeout | `(timeout)` |
| Do leave stage after timeout | `(timeout)A` |
| Don't leave stage after timeout | `(timeout)N` |

Table 8.1. Stage options notation in stage tree view

### 8.3.3 New Table…

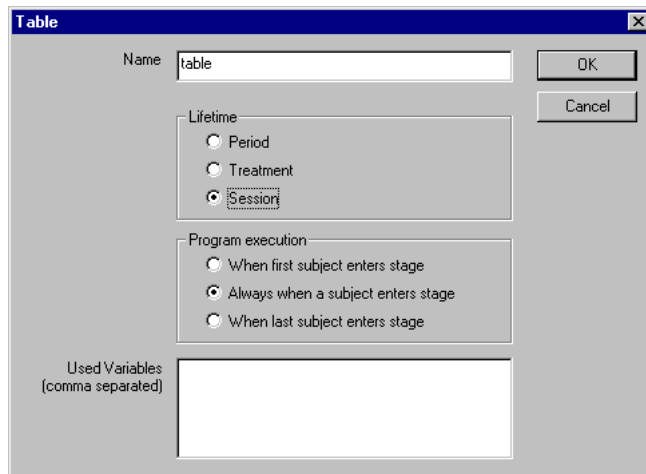A table stores input and calculations. Tables are accessed with their names.

Figure 8.2. Table dialog

### Name

Name of the table.

### Lifetime

How long the table can be used.

#### Period

The table is reconstructed after each period. As in the subjects or globals table, you get access to the data of the previous table with the prefix OLD in front of the table name.

#### Treatment

The table is available during the treatment. At the end of the treatment it is destroyed (after being stored of course).

#### Session

As is the case for the session table, the table survives the end of the treatment.

### Program execution

You can specify whether programs defined for the table at the beginning of a stage are executed when the first subject enters the stage, when the last enters or always when a subject enters that stage.

### Used Variables

For a table with lifetime session, you can define what variable are used. This allows them to be used in a program without having to define them within the treatment.

### 8.3.4 New Table Loader…

A table loader reads in a file that contains tables. The format corresponds to the z-Tree output file format. It is tab separated. In the first column there is the name of the table. Each table starts with a header line. The values of string variable have to be surrounded by quotes. Note that all variables have to be defined in advance. Otherwise, a error message is displayed and the corresponding columns are ignored.

Condition

> Condition whether the tables are loaded. The condition is evaluated in the globals table.

Append Filename

> Name of a file that contains tables. The records of these tables are appended to the existing records of these tables. An error message appears on z-Tree if the file cannot be found or opened.

Replace Filename

> Name of a file that contains tables. The records of these tables replace the existing records of these tables. Thus, the tables in the file replace the current tables. This is potentially dangerous if the data has not been stored. An error message appears on z-Tree if the file cannot be found or opened.

**Example of an input file**

```
tablename1 -> t1var1 -> t1stringvar2 -> t1var3
tablename1 -> 1 ->      "text" ->       3
tablename1 -> 4 ->      "hello" ->      6
tablename1 -> 7 ->      "world" ->      9
tablename2 -> t2var1 -> t2var2
tablename2 -> 11 ->     12
tablename2 -> 13 ->     14
tablename2 -> 15 ->     16
```

### 8.3.5 New Table Dumper…

A table dumper stores tables in the format as explained above in the Table Loader Section.

Condition

> Condition whether the tables are stored. The condition is evaluated in the globals table.

Output Filename

> Name of the file in which the files are stored. An error message appears on z-Tree if the file cannot be found or opened.

Tables

> Names of the tables that are stored. The tables have to be separated by semicolons.

### 8.3.6 New Program…

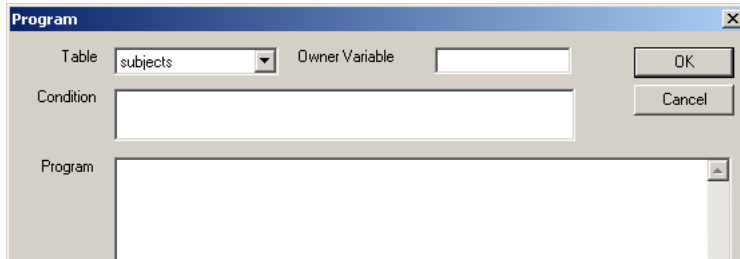In programs, calculations can be carried out. Programs modify the content of tables.



Figure 8.3. Program dialog

Table

The table in which the program is executed. For details see Scope environment.

Owner Variable

Allows access to a record in the subjects table with only the scope operator. It is the record whose value of the `Subject` variable equals the value of the owner variable in this table.

Example: You have a variable `Buyer` in the contracts table which contains the subject ID of the buyer. By putting `Buyer` into the Owner Variable of the program for the contracts table, you can access the record of the buyer in the subjects table just with the scope operator.

Condition

Condition for which records the program is executed.

This changes the scope environment in which the program is executed. See Section 9.2, "Scope environment".

Program

The program that is executed.

### 8.3.7 New External Program…

Starts a windows program, either on the z-Tree computer or on z-Leaf.

Condition

For z-Tree programs the condition is evaluated in the globals table and determines whether the program is started or not.

For z-Leaf programs, the condition is evaluated in the subjects table. If it is TRUE, the program is executed by the z-Leaf of the corresponding subject.

There is no error message if the program cannot be started.

## Run On

Defines whether the program is started on the z-Tree computer or on z-Leaf.

### z-Tree

The program is started on the z-Tree computer.

### z-Leaf

The program is started on the z-Leaf computers for which the condition is satisfied.

## Command line

Program name and parameters of the program as if they were entered in a command window.

## Current Directory

Directory in which the program is started. Be aware that the client computer might have a different file structure than the the z-Tree computer.

### 8.3.8 New Box

The subjects' screens are made up of boxes. These are rectangular parts of the screens. The boxes are positioned within the *remaining box* in the order in which they appear in the stage tree. At the beginning, the *remaining box* constitutes the whole screen. Later the *remaining box* is adjusted according to the definition of the boxes. The dialog for a box is explained below.
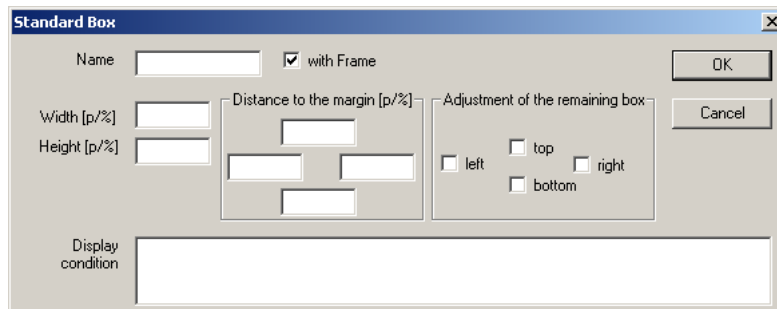


Figure 8.4. Common options for boxes

## Name

Name of the box. Used for documentation only.

## With frame

Whether there is a frame around the box or not.

## Width

Width of the remaining box in pixels [p] or percent [%]; optional.

Height

Height of the remaining box in pixels or in percent; optional.

Distance to the margin

Indicates how far away the margin is from the margin of the remaining box; in pixels or in percent of the remaining box; optional.

Adjustment of the remaining box

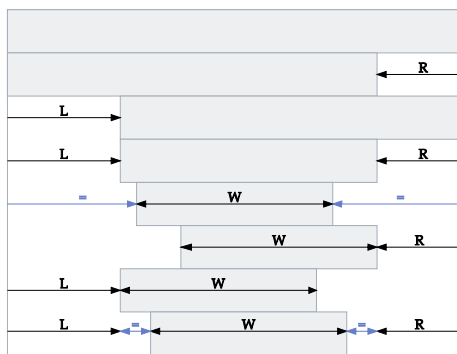Where is the remaining box cut-off, if at all?



Figure 8.5. Box placement within remaining box

Width, height and distance to the margin are optional. Depending on which fields have been filled, the box is placed within the remaining box. Figure 8.5, "Box placement within remaining box" shows all cases for the fields width (W), distance to left margin (L) and distance to right margin (R).

Display condition

If this field is nonempty, the box will only be shown if this condition is satisfied. The condition is calculated in the subjects table. It is reevaluated whenever data is changed, i.e. this condition can be used to turn on and off boxes dynamically. If the field is empty, the box will always be shown.

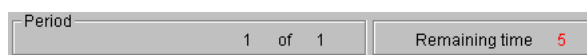**8.3.9 New Header Box…**

Creates a header box.



Figure 8.6. Header box with period and remaining time displayed on z-Leaf

Period number and time are displayed in a header box. The following options can be adjusted:

Show current period number

Period number is displayed or not.

Show total number of periods

Total number of periods is displayed or not.

Name of "Period"

Name in front of the period number. "Period" is used as standard. However, "Trading day" or "Round", for example, are also possible.

Term for "out of"

At "3 *out of* 10" (Periods).

Prefix for trial periods

"Trial" is standard.

Display time

Is the time being displayed to the subjects? If not, not even a time message is generated.

Term for "Remaining time"

"Remaining time [sec]:" is standard. If this field is empty, no time is displayed. In this way you can do without displaying the time and yet ask the subjects to continue when time has run out.

Term for "Please reach a decision"

Text that appears when the allotted time has run out. "Please reach a decision." is standard.

### 8.3.10 New Standard Box…
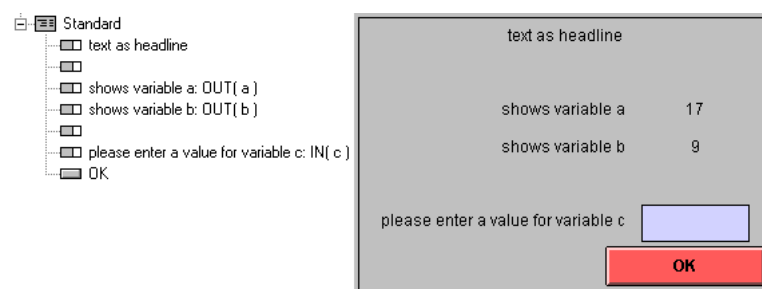
Creates a standard box.



Figure 8.7. Standard box in the stage tree and how it is presented on z-Leaf

Variables of the subjects table may be displayed or entered. The items are displayed from top to bottom. The window is divided into a label column (left) and a variable column (right). The variables are always displayed in the variable column. The label always appears in the left column if the variable is defined or if it consists only of an underscore _. If the variable is empty, the label is regarded as title and written centered over the whole window. If the label consists of more than one line, it is aligned to the left.

Button position

Buttons can be placed into corners or centered at a margin.

Button arrangement

If more than one button is defined in this box, the additional buttons can be placed either in rows or in columns. If they are placed *in rows*, z-Tree first tries to fill a row starting from the corner given by the button position. If the first row is full, a next row is filled starting from the same corner.

### 8.3.11 New Calculator Button Box…

Defines a button by which the Windows Calculator can be called up. It serves as a substitute for subjects who have no calculator.

Figure 8.8. Calculator button

### 8.3.12 New History Box…

Creates a history box. The results from previous periods are listed in a table. A label row contains the labels. If the table is too long, a scrollbar appears. The current period appears at the end of the table. The scrollbar is adjusted in such a way as to make this line visible at the beginning.

Figure 8.9. History box in the stage tree and how it is presented on z-Leaf

Show also current period

If checked all periods including the current period are shown. If it is not checked, only the previous periods are shown.

### 8.3.13 New Help Box…

Creates a help box. A help box is a box with a label that contains a text. A label and a help text can be entered. If the help text is too long to appear in the area reserved for the help box, a scrollbar appears.
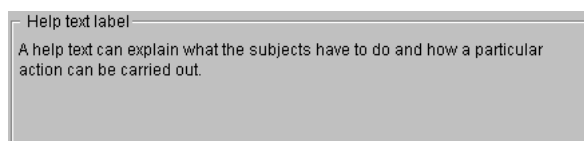
Figure 8.10. Client view of the help box

### 8.3.14 New Container Box…

Creates a container box, i.e., a box which may contain other boxes. If you want to directly place the boxes in the figure above, it becomes necessary to choose the appropriate sizes. By defining container boxes you can easily make sure that the boxes always match one another. Container boxes are boxes that can contain other boxes.
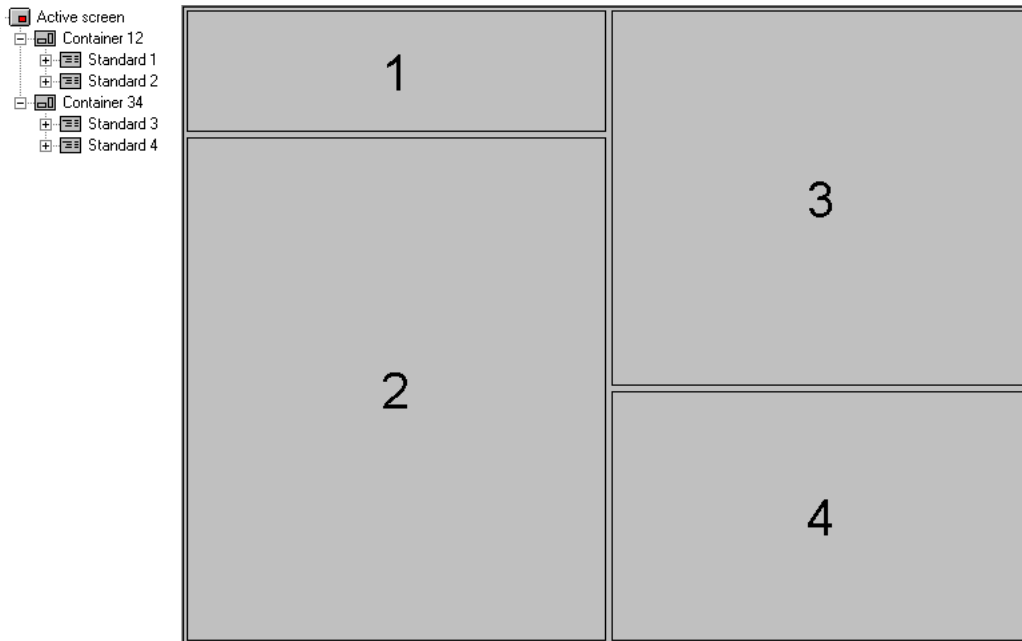


Figure 8.11. Container box in the stage tree and how it is presented on z-Leaf

In the example we define six boxes, the boxes 1,2,3, and 4, as well as two container boxes, 12 and 34. We define the width in the container box 12 and cut-off the remaining box at the left. Now container box 34 fills the region on the right. In boxes 1 and 3 we define the heights and cut the remaining box off at the top. In this way we have not entered a size specification at more than one place, and even if we should change something, the appearance of the screen remains the same.

If we move a box from one place to another inside the stage tree, it is moved *after* the box where the mouse button is released. By moving a box over the *icon* of a container box it is moved *into* the container box and positioned at its beginning.

### 8.3.15 New Grid Box…

Creates a grid box, a box that shows data of the subjects table in a tabular form.
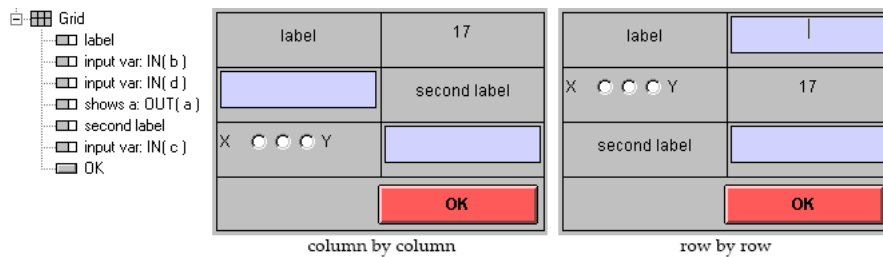
Figure 8.12. Grid box examples

The items are displayed in a table. Each item belongs to one field. If the item contains a variable, this variable is displayed. During checks the label is used to convey to the subject in which field he or she has made a mistake. The label is only displayed if the item does not contain a variable.

The individual options are:

Num. rows

Number of rows of the grid, including label row if it exists.

Num. columns

Number of columns of the grid, including label column if it exists.

Input row-by-row

The items are filled into the grid row-by-row. By pressing the tab key the subject can make the entry row-by-row.

Input column-by-column

The items are filled into the grid column-by-column. By pressing the tab key the subject can make the entry column-by-column.

First row contains labels

First row is a label row. Only used for *Separate labels by lines*.

Height [of the first line]

Height of the first line in % of the other lines.

First column contains labels

First column is a label column. Only used for *Separate labels by lines*.

Width [of the first column]

Width of the first column in % of the other columns.

### Separate labels by lines

If the first row is a label row, a dividing line is drawn between the first and the second row. The same procedure applies for the first column.

### Separate rows by lines

Lines are drawn between the rows.

### Separate columns by lines

Lines are drawn between the columns.

### Button position

Buttons can be placed into corners or centered at a margin.

### Button arrangement

If more than one button is defined in this box the buttons can be placed either in rows or in columns. If they are placed in rows, z-Tree first tries to fill a row starting from the corner given by the button position. If the first row is full, a next row is filled starting from the same corner.

**8.3.16 New Contract Creation Box…**

Creates a contract creation box. We show in an example how a subject can enter two new contracts in a contract creation box. A contract should contain the values x, y, and z. Therefore, we put three input items, x, y and z as well as an OK button into the contract creation box. Figure 8.13, "Contract creation box in the stage tree" shows how the contract creation box looks like in the stage tree.
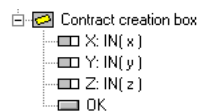


Figure 8.13. Contract creation box in the stage tree

How the contract creation box looks in z-Leaf can be modified in the box's dialog.
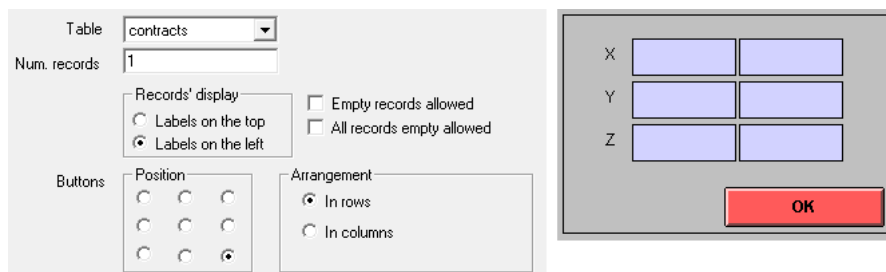


Figure 8.14. Contract creation box; records arranged with label(s) at the left hand side

Figure 8.15. Contract creation box; records arranged with label(s) on top

When you click the button, z-Tree checks if all entries have been made. Next the checkers in the button are checked, and if they are OK, two new contracts are added to the contracts table. Finally, the programs of the button are run. If we are in an auction stage, further entries can be made, otherwise the stage is concluded. You may specify the following options:

Table

All contract boxes can be linked to any user-defined tables.

Num. records

Number of records that have to be entered. Only one record is feasible in an auction stage. You can also enter a variable from the subjects table here, for example a constant, numContracts. In this way you can define treatments in which subjects have to enter different numbers of contracts.

Records' display

The records can be displayed in columns or in rows. In the first case the labels appear on the left hand side, in the latter case on top of the input fields. Figure 8.14, "Contract creation box; records arranged with label(s) at the left hand side" and Figure 8.15, "Contract creation box; records arranged with label(s) on top" illustrate the two options.

Empty records allowed

If this option is marked, the subject can fill out less records than are displayed. For each record, however, either all fields must be filled or none. In this way you can allow the subjects to offer less than the maximum possible number of contracts.

All records empty allowed

All records can be left empty which means that no record is created.

Button position

Position of the button (the buttons) in the box.

Button arrangement

When the first button lies in the lower right hand corner, horizontal means that further buttons are added on the left, until one has to begin a second row. The first button in the second row lies above the first button. The other cases are analogous.
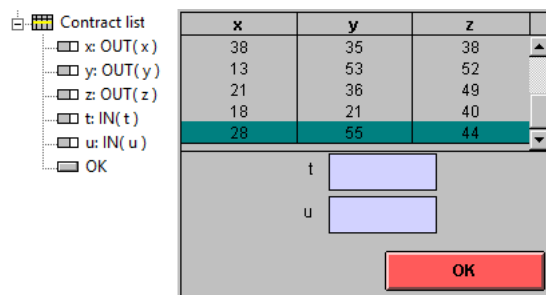
### 8.3.17 New Contract List Box…



Figure 8.16. Contract list box in the stage tree and how it is presented on z-Leaf

In a contract list box a selection of records is listed. Which records appear as well as the order of the records in the list can be set. Contract list boxes are continually updated. Therefore, whenever a subject effects a change in the contracts table, this is immediately registered and displayed in the contract list box.

The contract list box is composed as follows: The values of the output items are entered in one list, each record on a separate row. If there is not enough space for all contracts in the list, a scrollbar appears. If there are input items, they appear at the bottom of the list in the same layout as in a standard box.

When a contract list box contains a button, the subject can select a record and fill in the input items. When the button is clicked and the checks are OK, the variables of the input items are set and the programs are run for the selected record.

It is useful for the button of a contract list box to contain a program. Only with the help of a program can the contracts table be changed in such a way as to show that the contract has been selected and who has selected it. Such a program could be the following:

```
Accepter = :Subject;
```

`Accepter` is a variable of the selected records. `:Subject` is the variable `Subject` in its own record in the subjects table. (For details see Chapter 9, *Programming Environment* on programs below).

On the options:

Table

All contract boxes can be linked to any user-defined table.

Owner var.

Owner variable is a variable of the contracts table. The records for which the value of this variable equals the variable Subject are shown blue for each subject. Example: Seller is the variable in the contracts table in which the subject number of the seller is written. When all sales offers are displayed on the auction screens of the sellers, one can show one's "own" contracts in blue. These are the contracts for which the variable seller has the value Subject.

Condition

Condition that determines which records are to be displayed.

Sorting

Expressions according to which sorting is carried out, separated by semi-colons. When sorting is to be undertaken in descending order, the expression is preceded by a minus sign.

Example: `a; -b; -c;` sorts the columns in the following way:

| a | 1 | 3 | 6 | 6 | 6 | 7 | 8 | 8 | 8 | 10 | 10 |
|---|---|---|---|---|---|---|---|---|---|----|----|
| b | 9 | 2 | 7 | 5 | 5 | 4 | 4 | 4 | 2 | 6 | 3 |
| c | 7 | 2 | 4 | 5 | 3 | 2 | 8 | 4 | 5 | 3 | 3 |

Scrolling

After each change to the list scrolling to the beginning or to the end takes place.

Mark best foreign contract

If no other contract is selected, the contract in the direction of scrolling is automatically selected, i.e., if scrolling has taken place towards the end, the last foreign contract is chosen. A contract is considered as an *own* contract if the owner variable has the value of the subject ID of this subject. All other contracts are foreign contracts. If there is no owner variable, all contracts are considered as foreign.

Button position

Position of button (buttons) inside the box.

Button arrangement

When the first button lies in the lower right hand corner, horizontal means that further buttons are added at the left until a second row has to begin. The other cases are analogous.

## 8.3.18 New Contract Grid Box…
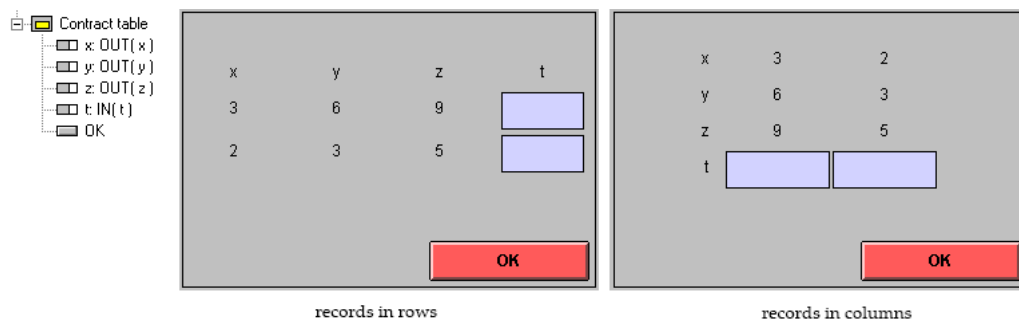


records in rows          records in columns

Figure 8.17. Contract grid box in the stage tree, and displayed with label on top and on the left hand side

The selected records are shown in a table. Values of the output items and input fields for the input items appear in this table. There must be space for all contracts in the table.

This box works similarly to the standard box, except that variables from the contracts table, not from the subjects table can be viewed and changed in it. The images above show contract grid boxes. On the left, the records are displayed in rows and on the right, in columns.

On the options:

Table

All contract boxes can be linked to any user-defined table.

Owner variable

The owner variable is a variable in the contracts table. For each subject, the records for which this variable has the same value as the variable Subject are shown in blue.

Condition

Condition for the records that are displayed. The expression in this field is evaluated for each record. Those records are shown for which the value of the condition is TRUE.

Sorting

Expressions according to which sorting is carried out, separated by a semi-colon. If sorting is to be carried out in descending order, the expression is preceded by a minus sign.

Records' display

As in the contract creation box.

Empty records allowed

If this option is marked, the subject can fill out less records than are displayed. For each record, however, either all fields must be filled or none.

All records empty allowed

All records can be left empty.

Button Position

Position of button(s) inside the box.

Button Arrangement

When the first button lies in the lower right hand corner, horizontal means that further buttons are added at the left until a second row has to begin. The other cases are analogous.

### 8.3.19 New Message Box…

Creates a message box. If there is a message box, messages generated by z-Tree are not displayed in separate windows. This makes the restart of a client easier because in case of a restart all messages reappear too and have to be confirmed with a mouse click.

Initial Message

> Appears when the stage starts.

### 8.3.20 New Multimedia Box…

In the multimedia box an image, a movie or sound file can be displayed. The content, i.e. the image, movie or sound of the multimedia box is stored in an external file. This file must be accessible at the client's computer. One way to achieve this is to put the files on a shared server and map this share to the same drive letter on each of the client computers.

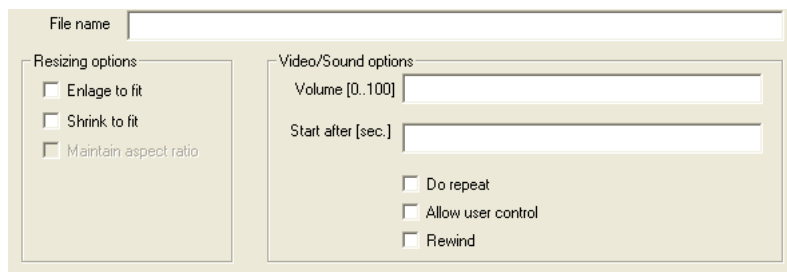Movies and sound are played as long as the box is visible. This allows for turning the media file on and off.



Figure 8.18. Multimedia box dialog

File name

> Name and path of the media file. Supported file types are jpg, gif, png, bmp, wmv, mpg, avi, wav, and mp3. The files are played using Microsoft Windows Media Player.

Enlarge to fit

> If the image is smaller than the box, it is enlarged.

Shrink to fit

> If the image is bigger than the box, it is scaled down.

Maintain aspect ratio

> If the image has to be stretched, it is stretched equally horizontally and vertically.

Volume [0..100]

> Volume of the sound. Default is 0.

Start after [sec.]

> Normally, movies and sound start immediately after the box is displayed, i.e. usually at the start of the stage. With this option the start of the movie or sound can be delayed.

Do repeat

The movie or sound is repeated indefinitely.

Allow user control

The user gets a control panel to move the movie or sound back and forth.

Rewind

If the box is turned off (hide), the movie or sound will play from the beginning when turned on again.

### 8.3.21 New Plot Box…

The plot box sets up a coordinate system and makes it possible to display *plot items*. Plot items are drawn sequentially, meaning that items lower down in the list of items can cover items higher up in the list. The Plot box is opaque and covers all boxes below it.
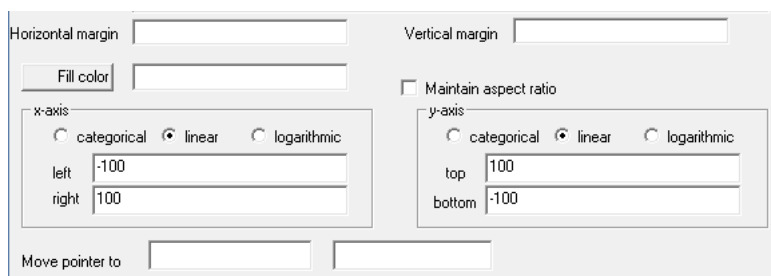


Figure 8.19. Plot box dialog

Horizontal margin

Distance from the frame of the plot box to the beginning of the plot area. Measured in pixels.

Vertical margin

Analogous.

Fill color

Color of the plot area.

Maintain aspect ratio

When checked, ensures that units horizontal and vertical are the same actual size, in case the z-Leaf screen is not square.

categorical

The extent of the range is increased by 0.5 in both directions. So categories 1 to 7 range from 0.5 to 7.5, which is a distance of 7. In addition, the labels on the axis are place within the ticks and not at the ticks.

linear

> Linear units define the coordinate system.

logarithmic

> Logarithmic units define the coordinate system.

left, right, top, bottom

> Determines the boundaries of the coordinate plane the plot box creates.

Move pointer to

> Whenever this box is displayed, e.g., when it is shown for the first time and when it is redisplayed after it has been hidden by setting the display condition to FALSE, the mouse pointer is moved to the corresponding x and y position. If both fields are empty, nothing happens. If only one of the two fields is nonempty, the mouse pointer is adjusted in the corresponding dimension. (Legal input: Expression in subjects table.)

### 8.3.22 New Chat Box…

The chat box allows subjects to enter text into an input field and show the text as output. Input is in text format, and is stored in a table between quotation marks and cannot be modified. The chat box is similar to a contract creation box, except that input is registered by hitting the enter key rather than with a button. Every time the enter key is pressed, a new record is added to the selected table, with the most recent text.
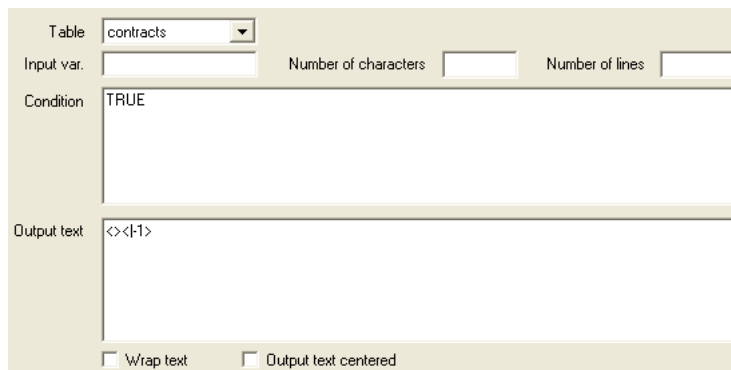


Figure 8.20. Chat box dialog

Table

> The chat box can be linked to any user-defined tables, as well as the contracts table.

Input var.

> The variable name for the chat texts.

### Number of characters

The number of characters a subject may enter before hitting the enter key, including spaces.
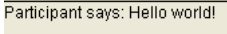
### Number of lines

The number of lines shown in the input text field.

### Condition

Condition that determines which records are to be displayed.

### Output text

Here you can determine what text appears before the chat text. For example, with what is written in the output text space on the left, the subjects will see what is written in the right hand area, when a subject types "Hello world!" into the chat entry area.

```
<>Participant says: <|-1>
```
Participant says: Hello world!

### Wrap text

Self-explanatory. If this is not selected, if there is text that extends beyond the screen width a scrollbar will appear.

### Output text centered

Output text is centered rather than the default left-aligned.

## 8.3.23 New On-Off Trigger…

The on-off trigger item can be placed into any box before other elements (e.g. items or buttons). As buttons and plot input items, it can contain checkers and programs. The starting scope of programs is the subject's record in the subjects table.
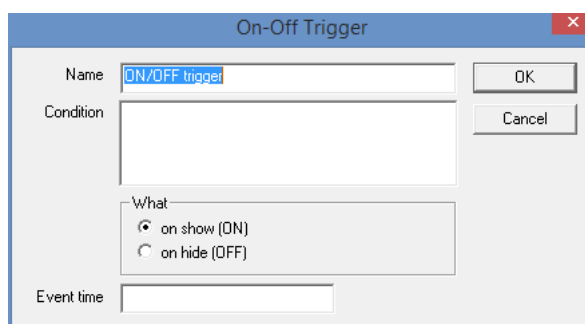


Figure 8.21. On-Off Trigger dialog

### Condition

The condition is evaluated on the client. It allows activating and deactivating the trigger.

### What

What kind of event triggers the action? Use on show (ON) if the appearance of the box should trigger the action and on hide (OFF) if the disappearance of the box shoud trigger the action.

### Event time

If this item is not empty then the corresponding variable is created in the scope of the corresponding box. This variable is filled with the time of when the box is shown or hidden, in seconds after the start of the period. Its accuracy is limited to the resolution of the system timer, which is typically in the range of 10 ms to 16 ms.

### 8.3.24 New Button…

Input is concluded by clicking a button. The labeling of the button can be defined. If the text consists of an underscore '_', no button appears. However, space for a button is kept free. If the button label contains the &-character, the next character is underlined. If you want to insert an ampersand '&' into the button label, you have to duplicate it.

Buttons can contain checkers and programs. Checkers are used for checking subjects' entries. Programs in buttons are executed when the button is clicked and when all checks are OK.

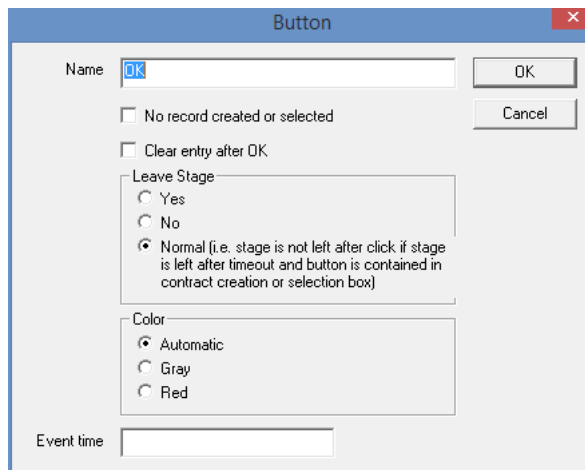Standard boxes, grid boxes and all contract boxes can contain buttons.



Figure 8.22. Button dialog

### Name

Label of the button. If this consists only of an underscore (_), no button appears. However, the space for the button is kept free.

### No record created or selected

If this option is checked for a button in a contract creation box, then pressing the button will not create a new record.

If a button in a contract list box has this option checked, then pressing the button will not select a record, i.e., this button can be pressed even if no record is selected.

### Clear entry after OK

After the button has been pressed, the fields belonging to this button are cleared. This option is useful to prevent subjects from making the same entry repeatedly.

### Leave Stage

Per default, stages are left, if the button is in a normal box or if the stage is not an auction, i.e., it waits for input to conclude. The options here allow overriding this behavior.

### Color

Buttons which are related to input appear in red other buttons in gray. This option allows more flexible use of the color.

### Event time

If this item is not empty then the corresponding variable is created in the scope of the item in which the button lies. This variable is filled with the time of when the event on the client occured, in seconds after the start of the period. Its accuracy is limited to the resolution of the system timer, which is typically in the range of 10 ms to 16 ms.

## 8.3.25 New Checker…

Checkers are used for more complicated input checks. If a condition is not met, a message is displayed.

### Condition

Condition that has to be satisfied. The message appears if the condition is not satisfied. In contract creation boxes, checkers are run for *each new* record. In contract list boxes, they are run for the *selected* record. In all other boxes, the check is carried out in the subjects table.

### Message

Message that appears if condition is not satisfied. If there is a message box, the message appears in this box. Otherwise, a dialog appears that needs to be confirmed by the subject.

### "yes"-Button

Accept the input nevertheless.

### "no"-Button

Do not accept input.

If there is only a "yes"-button, then the message appears and after pressing this button the input is accepted. If there is only a "no"-button, then the message appears and after pressing this button the input is rejected. If there is a "yes" AND a "no"-button, then the message should contain a question. Pressing the "yes"-button accepts the input pressing the "no"-button rejects it.

**Note on how checks are performed**

First of all we always check if each field is filled. The subject needs to enter a value into each field, unless it is explicitly stated in the item that the field may be empty.

A minimum value and a maximum value need to be entered in the definition of items. If a number has to be entered, the resolution is also given in the layout field. The subjects' entries must lie between the minimum and the maximum value and it must be a multiple of the resolution. If this is not the case, the entry is not accepted. If the label is not empty, the subject receives a message telling him/her which error has occurred in which field. If the label is empty, it bleeps and the field containing the error is marked. No message appears.

If all these checks are passed, the checkers in the button are handled.

### 8.3.26 New Item…

Items are used for displaying and reading in variables. An item contains the name of the variable and information on how to display it. We call an item "input item", if the checkbox *Input* is checked. In this case subjects have to make an entry. We call an item "output item", if the checkbox *Input* is not checked. In this case a variable will be displayed. Items can be contained in standard boxes, grid boxes, history boxes and all contract boxes.

Like buttons, items can contain checkers and programs. If this is the case, the item is called an active item. If the item contains checkers and/or programs, they are invoked when the state of the item is changed, for instance if a radio button is clicked or a slider is moved. (It does not work with text items, i.e. items in which a number or text is entered with the keyboard.)

Label

> Name of the variable displayed to the subjects. This text may also be empty.

Variable

> Variable that is displayed or value that is read in. In the case of input items only variables or array variables with a number as index are allowed. For output items any expression is allowed. If the variable is empty, the item only displays the label. If the variable contains only an underscore _, the layout is the same as when a variable has to be entered, except that no value for the variable is displayed.

Layout

> The layout specifies how variables are shown. For reals, a number specifies that the variable is rounded to a multiple of this number when it is displayed. The number can also be specified as a variable or an expression. If for instance 0.2 is entered in layout, and the variable contains the value 12.34, then 12.4 is displayed. If it is an input item, then the number is used to check whether the number entered by the subject is a multiple of this number. For string input, you write "!string". The minimum and maximum number of characters is specified in the minimum and maximum. For string output, the Layout is irrelevant. It can be left empty.

> An exclamation mark makes further formatting options available:

```
!text: value₁ = label₁; value₂ = label₂; …
```

In the case of an output item the text $label_i$ appears when the variable has the value $value_i$. In the case of an input item it is the other way round, i.e. if a text $label_i$ is entered, the variable gets the value $value_i$. If text different from all $label_i$s is entered, an error message appears. The label text has to be in single or double quote. You can use all types of variables for this option. Of course, all value specifications have to be from the same type.

```
!radio: value₁ = label₁; value₂ = label₂; …
```

Labeled radio buttons appear in a vertical order. The value of the variables again corresponds to the values, the labels of the buttons are the corresponding labels. You can use all types of variables for this option. Of course, all value specifications have to be from the same type.

```
!radiosequence: value₁ = label₁; value₂ = label₂; …
```

Labeled radio buttons appear in a horizontal order. The value of the variables again corresponds to the values, the labels of the buttons are the corresponding labels. You can use all types of variables for this option. Of course, all value specifications have to be from the same type.

```
!radioline: leftvalue = leftlabel; rightvalue = rightlabel; number
```

A row of radio buttons appears. They may be accompanied by labels on the left and on the right. Leftvalue is the value of the button at the left margin. Rightvalue is the value of the button at the right margin. The value of number determines the number of buttons. This option works only with variables of type real.

```
!slider: leftvalue = leftlabel; rightvalue = rightlabel; number
```

A slider appears. The slider may be accompanied by a text on the left and on the right. $leftlabel$ is the value of the variables at the left margin. $rightvalue$ is the value of the variables at the right margin. The value of number determines the resolution, i.e., the number of possible slider positions. This option works only with variables of type real.

```
!scrollbar: leftvalue = leftlabel; rightvalue = rightlabel; number
```

A scrollbar appears. The scrollbar may be accompanied by a text on the left and on the right. $leftvalue$ is the value of the variables at the left margin. $rightvalue$ is the value of the variables at the right margin. The $number$ sets the resolution, i.e., the number of possible scrollbar positions. This option works only with variables of type real.

```
!button: value₁ = label₁ ; value₂ = label₂; …
```

Buttons appear, arranged similarly to radio buttons. By clicking a button the variable is set to the corresponding value. Besides this, each button also has the same effect as the OK button. Hence, at most one variable per screen can have the button option. For output items, the button option yields the same display as the text option. You can use all types of variables for this option. Of course, all value specifications have to be from the same type.

```
!checkbox: 1 = text;
```

A checkbox appears. This checkbox is labeled with *text*. If the box is marked, it corresponds to the value `1`. If it is not marked, it corresponds to the value `0`. You can use all types of variables for this option.

```
!string
```

A box appears in which text can be entered. In contrast to the `!text` option, item input may be free text, and the input is stored in a string variable. The minimum and maximum field define the restrictions concerning the minimum and maximum number of characters that have to be entered. The `!string` option is more flexible than the `!text` option, allowing subjects to freely determine string variables. For displaying string variables, the `!string` option is not necessary. You just enter the number of characters that should be displayed per line.

> Some characters will be replaced in the output file: the quote is replaced by `\q`; new line by `\n`, tab by `\t` and back slash by `\\`.

### Input

If this checkbox is marked, the subject has to enter a variable. Otherwise the variable is displayed.

### Minimum

The minimum value permitted for input variables.

### Maximum

The maximum value permitted for input variables.

### Show value (value of variable or default)

This option appears only with input variables. Normally an input field is empty in the beginning. If this option is chosen, the input field is filled with the current value from the database.

### Empty allowed

If this option is marked, the variable may be left empty.

### Default

This option appears only with input variables. If a field is left empty, this value is taken as input instead.
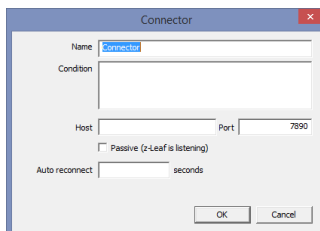
### Event time

If this item is not empty then the corresponding variable is created in the scope of the item. This variable is filled with the time of when the event on the client occured, in seconds after the start of the period. Its accuracy is limited to the resolution of the system timer, which is typically in the range of 10 ms to 16 ms.

## Examples of item layouts

| Layout | Input variable | Output variable |
|---|---|---|
| 2 | 6 | 6 |
| !text: 7 = "seven"; 8 = "eight"; 9 = "nine"; | seven | seven |
| !radio: 1 = "86.8"; 24 = "102.8"; | ⦿ 86.8<br>○ 102.8 | ⦿ 86.8<br>○ 102.8 |
| !radioline: 0="zero";5="five"; 6; | zero ○ ○ ⦿ ○ ○ five | zero ○ ○ ⦿ ○ ○ five |
| !radiosequence: 7="seven";8="eight";9="nine"; | ○ seven      ○ eight      ○ nine | ○ seven      ⦿ eight      ○ nine |
| !slider: 0 ="A"; 100= "B"; 101; | A ———————ʃ— B | A ———————ʃ— B |
| !scrollbar: 0="L";100= "R";101; | L ◄ ▯ ► R | L ◄ ▯ ► R |
| !checkbox:1="check me"; | ☑ check me | ☑ check me |
| !button: 1 = "accept"; 0 = "reject"; | accept<br>reject | accept |
| !string | | |
| 20 | | Hello World |

## 8.3.27 New Connector…



The connectors sets up a connection from a z-Leaf to another program. You can specify an active connection, in which z-Leaf connects to a listening port. Vice versa, in an passive connection, z-Leaf listens and the external program can connect. The connection is established the beginning of the respective stage or when the condition changes to TRUE. If an active connections breaks down, you can automatically reconnect. I can contain inlets and outlets. The are used to exchange information between z-Leaf and the host. Failed connections are reported in the logfile table.

Name

Name of the connector. Without consequence.

condition

Condition whether the connection should be active.

Host

IP address of the host that has to be contacted. If the field is empty, the local host is used. In order to base the host on a variable, you can use "<>".

Port

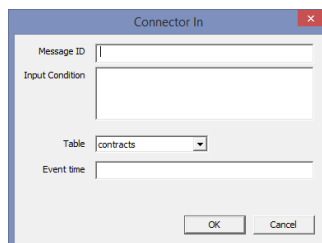Port under which the host has to be contacted.

Passive (z-Leaf is listening)

Not checked is active. In this case, z-Leaf connects to a listening port. Checked means passive. In this case, z-Leaf is listening.

Auto reconnect

Active connections can automatically reconnect when they break down. The time is specified in seconds. Decimal values are possible.

### 8.3.28 New Inlet…



The inlet handles messages sent from the host. You specify the table and the variables the date is read in. New records are created for data that comes in. You can put programs into an inlet. The programs are triggered when input arrives.

The exchanged data is text based. It is composed of lines, which contain tab separated information. Each line starts with the keyword `z-Leaf Connector`, followed by a message ID. Then, the variables follow, also separated by tab.
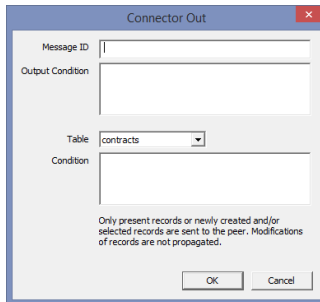
Message ID

Messages with this ID are treated by this inlet. Input Condition: Condition whether the inlet receives messages.

Table

Table, into which the data is written. Event time: This variable is filled with the time of when the a message is received, in seconds after the start of the period. Its accuracy is limited to the resolution of the system timer, which is typically in the range of 10 ms to 16 ms.

### 8.3.29 New Outlet…



The outlet sends messages to the host. You specify the table and the variables that are sent. When the Connector starts all records are transmitted. When a new record is created, the data of this record is sent.

The exchanged data is text based. It is composed of lines, which contain tab separated information. Each line starts with the keyword `z-Leaf Connector`, followed by a message ID. Then, the variables follow, also separated by tab.
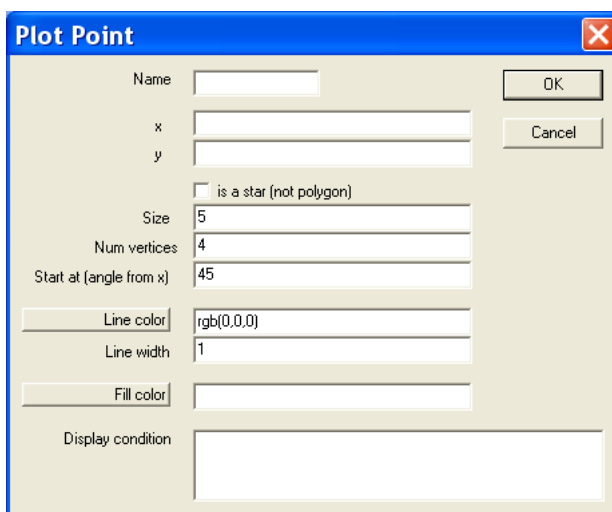
Message ID

Message that is sent as the ID, i.e., the second item in the message. Output Condition: Condition whether the outlet sends messages. This condition is evaluated in the subjects table.

Table

Table, from which the data is sent. Condition: Condition for whether a record is sent. The condition is evaluated in the record.

### 8.3.30 New Point…



x, y

Location of the center of the object.

is a star (not polygon)

Connects vertices to the center rather than adjacent to each other.



Figure 8.23. Example plot points

In the example on the right the checkbox *is a star* has been clicked, on the left it hast not.

Size

In pixels.

Num vertices

The greater the number of vertices, the rounder the object.

Start at (angle from x)

Angle at which a vertex of the polygon or the star is found, relative to the center.

Line color

In rgb.

Line width

In pixels.

Fill color

In rgb.

### 8.3.31 New Plot Input…

In a plot input, data can be modified when subjects make a new input, select an object, or drag an object. The action taken by the participant will trigger data to be modified (if specified). You can put programs into the plot inputs which will run when a subject performs the corresponding action. The trigger occurs when the mouse (or other trigger) is released. Locations all use the units determined in the plot box.
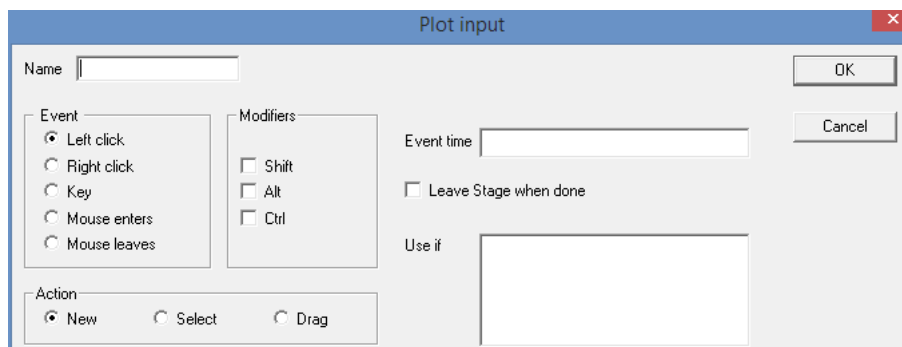


Figure 8.24. Plot input dialog

Event

What action taken by the participant should trigger an input.

Modifiers

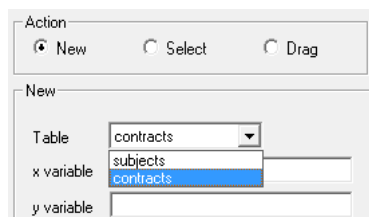What modifier should be used in order for the trigger to take place.

Action

What type of input action you are using (see below for further information).

Event time

If this item is not empty then the corresponding variable is created in the scope of the item in which
the plot input lies. This variable is filled with the time of when the event on the client occured, in
seconds after the start of the period. Its accuracy is limited to the resolution of the system timer,
which is typically in the range of 10 ms to 16 ms.

**Action New**

This plot input must be put inside a plot box. When a subject makes a new input (for example, clicking
somewhere on the plot screen), the variables named in the x/y variable area are overwritten with the
values of the location where the click was made.



Figure 8.25. Options for action new of a plot input

Table

In which table data should be modified.

x variable, y variable

The names of the variables to be modified.

**Action Select**

This plot input must be put inside an object (e.g. a rectangle). When the object is selected and released,
the center of the object is moved to the x/y location specified (unless left blank).

Figure 8.26. Options for action select of a plot input

x variable , y variable

The names of the variables to be modified.

**Action Drag**

An object can be dragged. To update an object's location, a program must be inserted in the plot input.



Figure 8.27. Options for action drag of a plot input

p0

Name of the variable to indicate the location where the object is first clicked.

p

Name of the variable to indicate the location of the mouse after clicking (but before release).

p'

Name of the variable to indicate where the object should be relative to p (the mouse location) as the object is being dragged.

Program to calculate p'

Where the object should appear relative to where the mouse is. This can either be the same as where the mouse is, or be restricted, for instance to allow the object to move in only one dimension (e.g. left and right). In the figure shown above, the object can be dragged like a slider, so only horizontally. The p' variable y2 is always zero, whereas the p' variable x2 equals x1, so the object will follow the mouse left and right but not up and down.
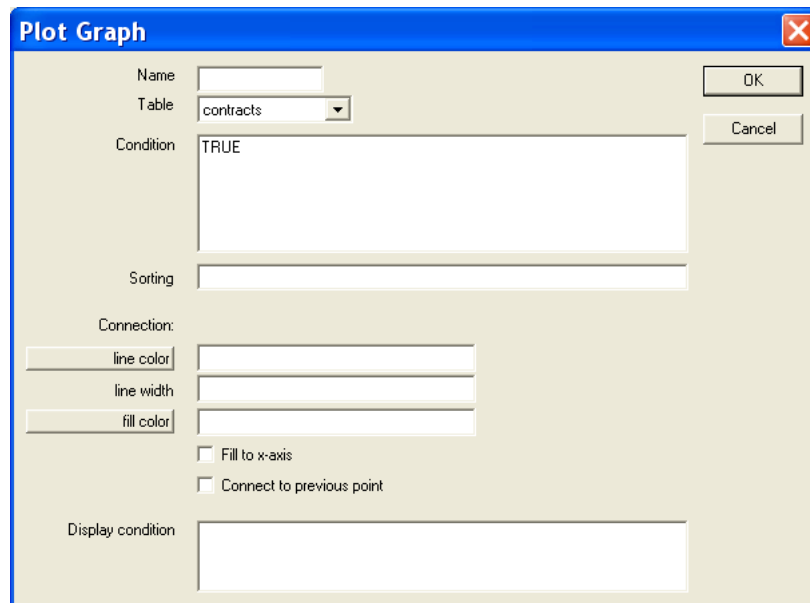
### 8.3.32 New Plot Graph…



Figure 8.28. Plot graph dialog

Table

Any custom table as well as the contracts table may be used.

Condition

Which records should be shown in the graph.

Sorting

Expressions according to which sorting is carried out, separated by semi-colons. When sorting is to be undertaken in descending order, the expression is preceded by a minus sign.

Line color

Used to connect points in the plot graph.

Line width

Used to connect points in the plot graph.

Fill color

Used to fill the polygons that are defined by points that are contained in the plot graph.

Fill to x-axis

Instead of filling polygons, it creates an area plot, i.e. it fills the area between the points that are contained in the plot graph and the x-axis.

Connect to previous point

Whether the different records in the given table should be connected by a line as they are represented on the screen.
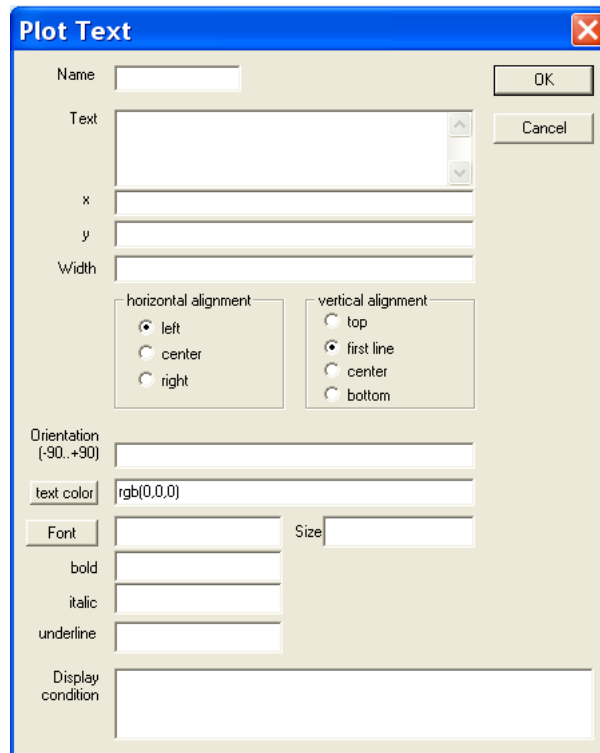
### 8.3.33 New Plot Text…



Figure 8.29. Plot text dialog

x, y

Where the text is to be located.

Width

Width of the box containing the box.

horizontal alignment, vertical alignment

How the text is aligned in relation to x/y.

Orientation (-90..+90)

Rotates text.

text color

In rgb.

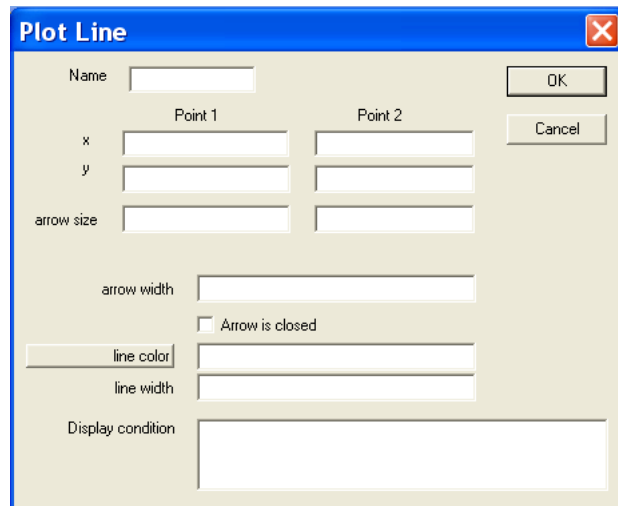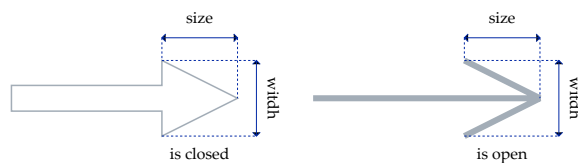Font

In pixels.

### 8.3.34 New Line…



Figure 8.30. Plot line dialog

Point 1 , Point 2

x and y locations of the two points to be connected.

arrow size , arrow width , Arrow is closed
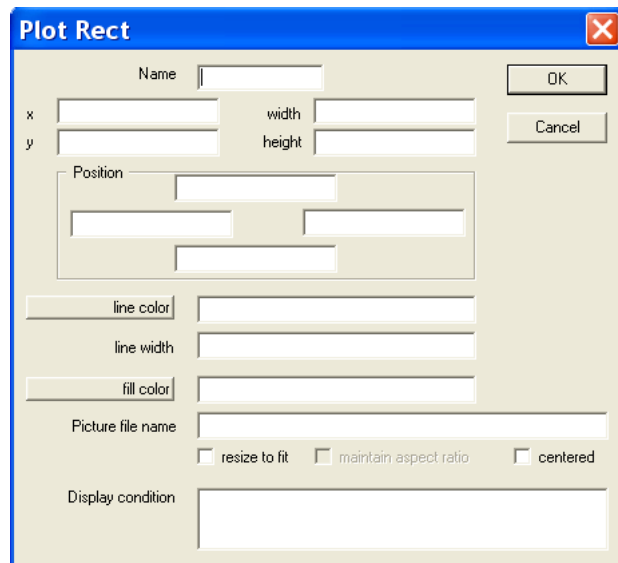
### 8.3.35 New Rect…



Figure 8.31. Plot rect dialog

x, y

Location of the object.

width, height

Size of the rectangle in units defined in the plot box.

Position

Relation of the beginning of the respective edge of the object to the location shown in x/y coordinates, in units determined in the plot box. If left blank, the rectangle is centered at the specified location.

Picture file name

Path and name of the picture file to be shown in the rectangle. Only bmp files are supported.

resize to fit

Check this box to fit the picture into the size of the rectangle.

maintain aspect ratio

Check this box to ensure the picture is not distorted from its original proportions.

centered

Position picture in the center.
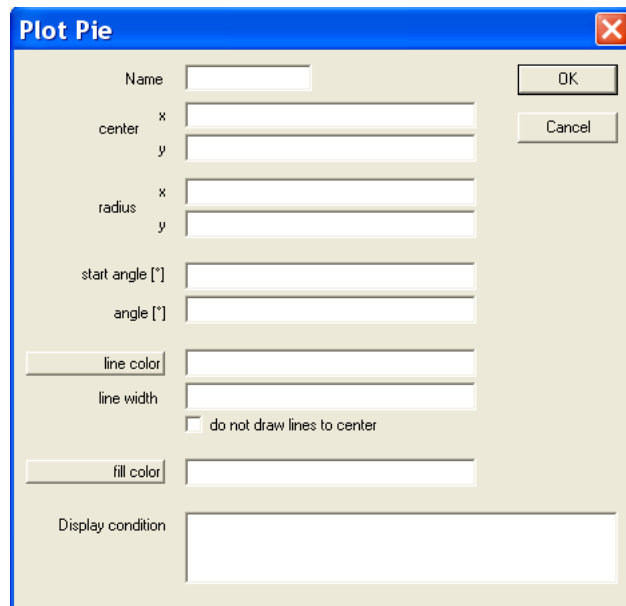
### 8.3.36 New Pie…



Figure 8.32. Plot pie dialog

Draws a circle, an ellipse or a pie as in a pie chart.

center

> Location of the vertex of the pie (center if it is a circle or ellipse).

radius

> Vertical distance (x) and horizontal distance (y) from the center to the edge of the pie.

start angle

> The angle measure at which the pie begins, measured in degrees, counterclockwise from the positive x-axis.

angle

> Size of the angle, measured in degrees.

do not draw lines to center

> If this item is checked, the outline consists only of the bow and not of of the line segments that connect to the center. The outline is drawn if line color and size are specified.

### 8.3.37 New Axis…
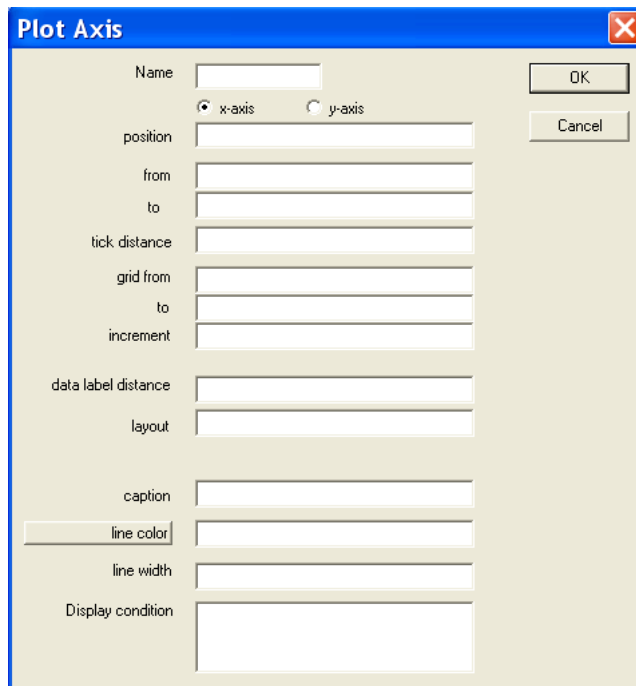


Figure 8.33. Plot axis dialog

Name

> Name of the axis. The name has no consequence and can serve as a comment.

x-axis, y-axis

> Defines whether the axis is horizontal (x-axis) or vertical (y-axis).

position

> Where the axis goes through. For an x-axis this is a y value and vice versa.

from, to

> For an x-axis it defines left and right borders; for a y-axis it defines bottom and top border.

tick distance

> Distance between ticks. The first tick is drawn at the from-position. If the tick should start in the middle, use two axes and put the ticks only on the narrower axis.

grid from, to, increment

> Allows to draw a set of lines vertically to the axis.

data label distance, layout

> Distance between labels that display the value of the axis. The layout has the same format as the item. Whether the data label is positioned at the tick or between the ticks depends on how the axis is specified in the plot box containing this axis item. If the axis is specified as linear or logarithmic the data label is positioned at the tick, and if the axis is specified as categorical it is positioned between the ticks.

captions

> Text that is displayed close to the axis. If you do not like the placing, use a plot text instead.

line color, line width

> Color and width of all lines of an axis. This includes ticks and grid lines. Use more than one plot axis if you want to differentiate.

Display condition

> Condition whether the axis is shown or not.

### 8.3.38 New Slide Show…

A slide show is a timed display of plot boxes. It has to contain slides or slide sequences. The plot boxes have to be contained in slides.

Skip delayed slides

> Checked if checked slides have to be omitted if it took too much time to process the previous slide(s). If slides have to be omitted, a message is reported in the logfile table.

Synchronize with screen refresh rate (needs DirectX)

> Allows to synchronize the presentation of the slides with the screen refresh rate. It requires that DirectX is installed. This can be used for subliminal priming. I this option is checked, then the duration of slides on the slide dialog is interpreted as number of frames.

### 8.3.39 Slide sequence…

Defines a sequence of slides based on the records of a table. Variables in elements within the slide sequence have access to the variables of the records of the slide sequence. As plot graphs, slide sequences can be nested. If nested, the scope environment is set up as in the case of the plot graph.

Name

> Name of the slide sequence.

Table

> Any custom table as well as the contracts table may be used.

Condition

Which records should be shown in the slide sequence.

Sorting

Expressions according to which sorting is carried out, separated by semi-colons. When sorting is to be undertaken in descending order, the expression is preceded by a minus sign.

Display Condition

Whether the whole slide sequence is shown or not.

Repeated

If the slide sequence should be repeated then add a value larger than 0. The sequence is shown 1 + this number times.

### 8.3.40 New Slide…

Defines a single slide. The slide can contain plot boxes.

Name

Name of the slide.

Display Condition

Whether the slide is shown or not.

Duration

How long is the slide shown, i.e. when the next slide is started to be displayed. The unit depends on whether Synchronize with screen refresh rate in the slide show is checked or not. If it is not checked, the time is in seconds. If it is checked, the time is in screen refresh frames, which is usually around 16 ms.

### 8.3.41 Expand All

Shows all elements by expanding any nested elements, i.e. elements contained within other elements.

### 8.3.42 Stage Tree

Opens the window with the stage tree. This menu is only active when the parameter table window is in front.

### 8.3.43 Parameter Table

Opens the window with the stage tree. This menu is only active when the stage tree window is in front.

**8.3.44 Check**

Checks the programs and parameters for errors.

**8.3.45 Import Variable Table…**

You give the name $v$ of a variable and choose a text file that contains a table. Then the line

```
v = w;
```

is added into the program of the specific parameters. Here $w$ stands for the value from the table. If the format of the table does not correspond with the format of the parameter table, the specific parameters are filled where there is a value in the table. This menu is only active when the parameter table window is in front.

**8.3.46 Show Variable…**

Sets the name of a specific parameter to the value of a variable. Then this value becomes visible in the parameter table. This menu is only active when the parameter table window is in front.

**8.3.47 Append Variable…**

Append the value of a variable to the name of a specific parameter. This menu is only active when the parameter table window is in front.

**8.3.48 Append Text…**

Append some text to the name of a specific parameter. This menu is only active when the parameter table window is in front.

**8.3.49 Matching**

**Partner**

A partner design for group matching is selected: The first subjects constitute group 1, the next, group 2, etc., for each period. If a range of the parameter table is selected, group matching is only carried out for this region.

**As First Selected Period**

Groups are matched for all periods in the same way as they are matched for the first period. If a region of the parameter table is selected, group matching is only carried out for this period. The first period of the selection counts as the first period.

**Stranger**

Group matching is carried out at random for each period. If a region of the parameter table is selected, group matching is only carried out for this region.

**Perfect Stranger**

Group matching is carried out in such a way that each subject is never in the same group with any other subject more than once. This is only possible for a limited number of periods. In the periods in which such a group matching is no longer possible, 1 is set as group number for all subjects. If a region of the parameter table is selected, group matching is only carried out for this region.

The execution of this command can take up some time. Depending on the given number it can require milliseconds, minutes or days. Before the command is executed, a warning appears and you get the possibility to cancel the execution of the command.

**Typed Perfect Stranger**

This creates an perfect stranger matching for heterogeneous groups: If, for example, a group consists of a company and two employees, we often do not want the same subject to have different roles. This means that only those groups that contain a company and two employees are allowed.

The execution of this command can take up some time. Depending on the given number it can require milliseconds, minutes or days. Before the command is executed, a warning appears and you get the possibility to cancel the execution of the command.

In the following table the number of periods for which we have calculated typed perfect stranger is listed. The values for which it is known that it is the highest possible value are given in boldface.



Figure 8.34. typed perfect stranger matching

**Transform**

Allows making transformations on the matching. There are three options: `Add`:: You can add a fixed number to the group ID. `Multiply`:: You can multiply the group ID with a fixed number. `Shuffle`:: You can shuffle the group IDs within the period.

With this command one can for instance easily define matching groups. Using matching groups is an important experimental technique because in a stranger matching there is generally only one independent observation per session. With matching groups one can define a stranger matching with more than one independent observation. For that, the subjects are divided into two (or more) sub-populations. Subjects are only matched within these sub-populations. Therefore these sub-populations constitute independent observations.

**Example**

You have 24 subjects with groups of 3 and you want to make two matching groups of 12 subjects each. You have to do the following:

1. Set the number of groups to 4.

2. Select the first 12 subjects in the parameter table and choose Treatment → Matching → Stranger.

3. Select the last 12 subjects in the parameter table and choose Treatment → Matching → Stranger again.

4. Choose (the last 12 subjects are still selected) Treatment → Matching → Transform…. In the dialog that appears select the radio button `Add` and enter `4` into the `Value` field.

5. Set the number of groups to 8.

### 8.3.50 Utilities

Treatment → Utilities → Make Stage Names Unique changes the names of the stages by adding numbers in such a way that the names of the stages are unique. This is important because the state of the clients is defined by the stage name.

### 8.3.51 Language

Sets the language for elements that are *newly* created as header boxes etc. All texts within these elements can be modified later, i.e. they can also be translated later.

## 8.4 Questionnaire Menu

### 8.4.1 Info…

Opens a dialog in which the parameters of the selected element can be viewed and changed.

### 8.4.2 New Address Form

The address prompt asks for the subject's address. The fields First Name, Last Name and Continue (button label) are compulsory if the address form should appear. The other fields are optional. If the question texts in the optional fields are left empty, the corresponding question does not appear on the clients' screens. If the fields First Name and Last Name are left empty, then no form appears. The payment file is written whenever the last subject has completed (or passed through without completing) the address form. Additional variables from the `session` table are added to the address form if questions are placed into the address form.

### 8.4.3 New Question Form

Question forms consist of a list of questions. The layout of the question is adjusted with rulers. All question forms except the last must contain a button.

Name (ID)

Name of the form. The name must be unique within a questionnaire.

Title

Title as it appears on the clients' screens.

Program

A program that is conducted in the session table for the subject who enters this questionnaire. If the variable `Participate` is set to zero, the questionnaire can be omitted.

### 8.4.4 New Ruler

Rulers are used to set the regions where labels and questions are positioned.

Distance to left margin

The position can be given in points or in percentage of the screen width.

Distance to right margin

The position can be given in points or in percentage of the screen width.

Distance between label and item

This is the horizontal distance between the label and the question. Half of this distance is also kept to the left and right margins.

Size of label

This is the maximum width of the label. When the display is given in a percentage, this refers to the width between the left and right margin and not to the width of the whole screen.

### 8.4.5 New Question

Questions in questionnaires correspond to items in treatments. In items, display options are defined in the field layout. Here they can be set with radio buttons. The fields are as follows:

Label

>	Name of the question as it is shown to the subjects. If no variable is set, the label appears as text over the whole width of the screen.

Variable

>	If it is an input variable, this is the name of the question as it should appear in the data file. If input is not set, it has to be a variable of the `session` table.

Type

>	Layout of the question.

>	Text, Number

>>		A text field appears into which input is entered. In the case of the number option there is a check to see if it is really a number that has been entered. Furthermore, the check determines whether the number is in the valid range. If the option Wide is selected for a text field, the entry field may consist of several lines. The number of lines can be selected.

>	Buttons

>>		A button is created for every line in the field Options. Of course, only one question with this option is permitted per question form.

>	Radiobuttons

>>		A radio button is created for every line in the field options. These buttons are arranged vertically one on top of the other.

>	Slider, Scrollbar

>>		The slider and scrollbar make a quasicontinuous entry possible. Minimum, maximum, and resolution need to be entered. In the wide layout, a label can be placed at the margins. These labels are entered in the field options. The maximum is always the value at the right border, the minimum is the value at the left border. Maximum may therefore be smaller than minimum.

>	Radioline

>>		A radio button appears for all possible answers determined by minimum, maximum and resolution. In the wide layout, a label can be placed at the margins. These labels are entered in the field Options. The middle button can be separated from the others by entering a number greater than zero for Distance of central button. The maximum is always the value at the right most button, the minimum is the value at the left most button. Maximum may therefore be smaller than minimum.

Radiolinelabel

Label line for radio button lines. Labels can be placed above the buttons at the margins. The texts for this action are in the Options field.

Checkbox

A checkbox is created for every line in the Options field. These checkboxes are arranged vertically one on top of the other. The resulting value is a list of all options checked.

Wide

In the wide layout, the entry region for the question appears under the label over the whole width and does not only appear on the right of the label in a second column.

Input

This field determines whether a question is presented to the subjects (input is set) or whether a variable is only shown (input is not set).

Empty allowed

This is set when the question does not need to be answered.

Minimum, Maximum, Resolution

When numbers have to be entered, these values are used for checking. With sliders, scrollbars and radio lines these values are used for converting the position into a number.

Num. rows

In the wide layout for text entries, this is the number of rows that can be entered.

Options

The labels of buttons, radio buttons and, in the wide layout, of sliders, scrollbars and radio lines.
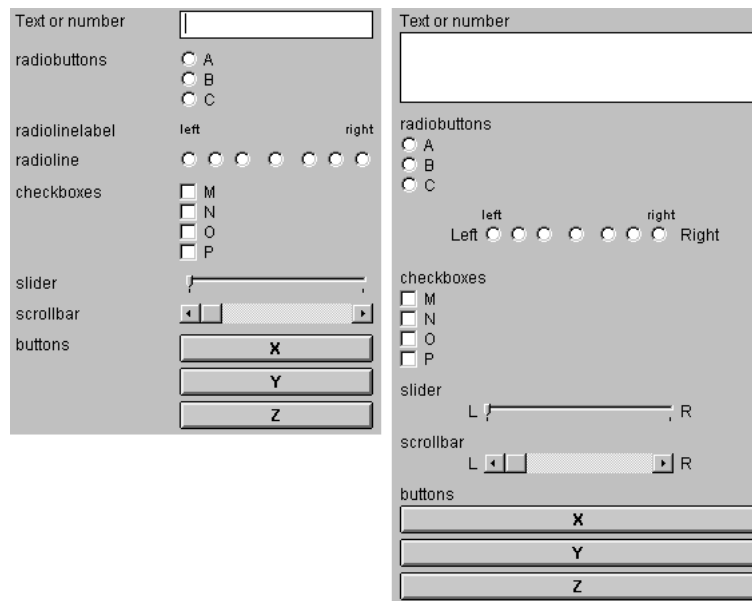
Figure 8.35. Layout of questions, normal layout (left) and wide layout (right)

### 8.4.6 New Button

Buttons conclude question forms. You cannot define more than one button per form. However, this button can be positioned anywhere on the form. The label of a button can be selected at random. All but the last form of a questionnaire must have a button.

### 8.4.7 Expand all

Shows all elements by expanding any nested elements, i.e. elements contained within other elements.

### 8.4.8 Check

Checks the questionnaire.

### 8.4.9 Language

Sets the language for elements that are newly created as address forms. All texts within these elements can be modified later, i.e. they can also be translated later.

### 8.5 Run Menu

### 8.5.1 Clients' Table

Opens the *clients' table*. The clients' table shows which clients are connected to the server. It also contains information about the state of the clients, i.e., which screens are currently being displayed to the clients. You may open the clients' table with the first command from the Run menu.

Each line corresponds to a subject. This subject is seated at a PC on which the client is running (column 'Clients'). As long as no treatment is started, the clients can be moved. As soon as you start a treatment, the order of the clients is fixed. After a treatment has started, only those clients that are not taking part in the current session can be moved. These clients are listed at the end of the table. So long as no treatment has been started, the clients can be sorted and shuffled with the commands Run → Sort Clients and Run → Shuffle Clients respectively. The clients' table has five columns:

Client

> Name of the client. In general this is the name of the subject PC. If a client is no longer connected, its name appears in brackets. If a client with the same name should reconnect, it can continue at the same place where the previous subject left off. The caption to the clients' table gives the number of clients currently connected.

State

> The state in which the subject is at a given point. Example: `Ready` when the subject is waiting for the next treatment or for the next questionnaire. The state is `*** abc ***` if the subject is in the active screen of stage abc and `- abc -` if the subject is in the waiting screen of stage abc.

Profit

> Accumulated income made in the previous treatments. The profit made in the current treatment is **not** included. This profit is indicated in monetary units (Fr., €, $, £, …), as various treatments may have different exchange rates.

Show up fee invested

> `TRUE` if the subject has suffered losses and has declared himself ready to account for these with his/her show-up fee.

Money added

> Extra money that the subject has injected into the game after losses in order to be able to continue. This sum is of importance for the calculation of losses. The profit in the payment file is always the net income the subject has gained from the experiment. At the hour of payment, injected money must therefore be added to the amount indicated in the payment file.

### 8.5.2 Shuffle Clients

Shuffles the clients at random.

### 8.5.3 Sort Clients

Sorts the clients. Should names have the same beginning and a number at the end, they are sorted according to the value of the number at the end, i.e., b2 comes before b11.

### 8.5.4 Save Client Order

Saves the order of the clients and how the clients can be reached over the network in order to restart them after a crash of z-Tree. When the first treatment is started, this is done automatically. The command is very useful if you have a non-arbitrary assignment of types to clients. It is a good practice to execute the command after the clients have been connected.

### 8.5.5 Start Treatment

Starts the treatment that is in the front window as soon as possible. If it is the first treatment, z-Tree first checks whether the correct number of clients is connected. If not, a warning appears and the start can be canceled. If the treatment is allowed to start, z-Tree waits until at least as many clients are connected with the server as are needed to run the treatment. Then the treatment is started. If it is not the first treatment, the treatment starts immediately.

### 8.5.6 Start Questionnaire

Starts the questionnaire, if this is possible. A treatment needs to have been started beforehand and the clients need to be in the state Ready.

### 8.5.7 *tablename* Table

Opens the window showing the table or brings it into the foreground.

### 8.5.8 Stop Clock

Stops the clock. However, subjects can still make entries. If, for instance, all subjects click continue, the session continues regardless of the stopped clock.

### 8.5.9 Restart Clock

The clock is restarted.

### 8.5.10 Discard a client

The client selected is discarded. It can be replaced by a new client. In case of a defect in a subject PC, the following needs to be done: The client in question is clicked in the clients' window. Then the menu Discard a client is chosen. With this the client is discarded. The new PC is started. The client then appears in the lowermost row of the clients' window. It first has to be selected and then moved over the old client. Doing this the mouse has to be released for an instant. If the mouse is not released this procedure only selects the space between the new and the old client.

### 8.5.11 Leave Stage

The subject whose state is selected leaves the stage: If a client is in an active screen, it can be moved on to the waiting screen of the stage. This option is useful when testing program. In an ordinary session it

should not be necessary to use this command. Note that you have to select the subject's state in the clients' table.

There is the variable `LeaveStage` which does the same from within a program. When this variable is set to 1, the client goes to the waiting state of the current stage.

### 8.5.12 Stop after this period

When the period that is running is finished, the treatment is stopped. When you have applied the command, a check mark is shown in the menu. You can undo it by applying it again (scroll down to it and click on it). Then the check mark disappears. This command is particularly useful when testing a treatment.

### 8.5.13 Restart all clients

All clients are restarted. For this z-Tree has to know the clients' addresses and the z-Leaves must be accessible over the internet (for instance, not behind a firewall). The addresses of the leaves are stored when the first treatment starts and when the command Run → Save Client Order is applied.

### 8.5.14 Restore Client Order

Restore the clients' order as it was within the previous session or after the last execution of the command Run → Save Client Order.

### 8.5.15 Reload Database

Reloads the database, i.e., it reloads all tables. This command is used after a crashed session.

### 8.6 Tools Menu

### 8.6.1 Separate Tables…

Reads a xls-file that z-Tree generated and creates new files: One file per table and treatment. You can select more than one file at once.



The output files will overwrite existing ones. Because the names generated are based on the original file names this should not cause problems. You cannot read in the file of the running session. Close z-Tree before separating the tables. Do not use the tools while a session is running.

### 8.6.2 Join *.sbj file…

Joins the sbj-file to a file that contains a table with a column named 'Subject' (input table). Joining means that the information in the sbj-file, i.e., the questionnaire data, is added to each row of the input table. Which line of the sbj-file is entered depends on the value in the 'Subject' column.

Note that the sbj-file is transposed with respect to the xls-file.

The output file will override an existing one. Because the names generated are based on the original file names that should not cause problems. You cannot read in the file of the running session. Close z-Tree before separating the tables. Do not use the tools while a session is running.

### 8.6.3 Join files…

It allows you to add the lines of one table to the lines of another table based on key variables. For instance, you can add the data in the subjects table (key table) to the data of the contracts table based on the key ('SessionID', 'TreatmentNumber', 'Period' ,'Subject').

Data File

Path of the data file. If you click the label Data File, a standard file-open dialog opens. To each row of the data file, some row of the key file will be added.

Key File

Path of the key file. If you click the label Key File, a standard file-open dialog opens. The key file contains the information to be added to the data file.

Link variable(s)

comma-separated list of variables in the data file.

Key variable(s)

comma-separated list of variables in the key file.

Output file

Path of the output file. If you click the label output file, a standard save-file dialog opens. The data file contains one row per row of the data file. Each row contains the corresponding row of the data file followed by a row of the key file. The row that is taken from the key file is the row in which the key variable has the same entry as the link variable in the data file.

**Example**

Data file:

| a | b | c |
|---|---|---|
| 4 | 3 | 7 |
| 5 | 1 | 8 |
| 6 | 1 | 9 |

Key file:

| k | d |
|---|---|
| 1 | 11 |
| 2 | 22 |
| 3 | 33 |

Data variable: b, Key variable: k

Output file:

| a | b | c | k | d |
|---|---|---|---|---|
| 4 | 3 | 7 | 3 | 33 |
| 5 | 1 | 8 | 1 | 11 |
| 6 | 1 | 9 | 1 | 11 |

### 8.6.4 Append Files…

Several files containing tab-separated tables with a header (called input tables) can be merged: The output table is a table containing all the records of all the input tables. If a column does not exist in some of the input tables, the value is left empty in the rows of these tables.

> The output file will override an existing one. Because the names generated are based on the original file names that should not cause problems. The tools cannot be used for modifying the files of the running session. Do not use the tools while a session is running.

### 8.6.5 Split Files…

Splits a table into parts that can be read by old versions of Excel. Should be obsolete.

### 8.6.6 Expand Timefile…

Adds a data file to the time file that has been generated with the `/logtime` command line option in z-Leaf.

### 8.6.7 Fix File…

If there is a severe problem in z-Tree, and in some sessions the data is not correctly logged in the xls-file, what happens? In tables which do not have a fixed number of records (the contracts table or user defined tables), it can happen that records are stored more than once. This occurs only if there are fewer records in a later period than in an earlier period. In this case it occasionally happens that the table is 'filled' with old data.

Fortunately, this error can easily be recognized in the file. The command Tools → Fix File… corrects a corrupt file. It also gives you a message whether there was actually an error in the file. Apply this command to the xls-files of sessions where you worked with the contracts table or with user defined tables that did not have a predefined number of records.

> A test for this error is also included in the command Tools → Separate Tables…. There is no problem if you use only the globals and the subjects table.

### 8.7 View Menu

### 8.7.1 Treatment and questionnaire files

All treatments and questionnaires opened in z-Tree can be brought into the foreground in the windows menu.

### 8.7.2 Toolbar

Opens and closes the toolbar.

### 8.7.3 Status Bar

Opens and closes the status bar.

### 8.8 Help Menu

Not yet implemented.

# 9 Programming Environment

## 9.1 Programming language

### 9.1.1 Error handling

There is no exception handling in z-Tree. You have to make sure that the parameters of a function and the arguments of operators are valid. Illegal values produce unpredictable results and may cause z-Tree to crash.

### 9.1.2 Comments

There are two forms of comments. The end line comment starts with `//` and ends at the end of the line. The other comment form starts with `/*` and ends with `*/`. Comments cannot be nested but the different forms of comments ignore each other. So, if you use end line comments to comment your statements, you can comment out the statements using the `/* ... */` comment.

### 9.1.3 Variable types

There are four basic types of variables; real, string, boolean, and color.

– `real` variables contain numbers. The numbers use 8 bytes. They have about 15 decimal digits with a range of approximately 5.0e−345 to 1.7e308.

– `string` variables contain text. They are defined using quotes (see below).

– `boolean` variables contain the result of a condition. They are either TRUE or FALSE.

– `color` variables contain a color. Internally, a color is represented with the three values for red, green, and blue. The values for red, green, and blue can be given as numbers between 0 and 1 as in `rgb(1, 0, 0.5)` or as integer numbers between 0 an 255 as in `rgb255(255, 0, 128)`.

Variables have a fixed type. If you try to assign a variable (or a function) of a type to a variable of another type, you will get an error message.

### 9.1.4 Constants

```
TRUE
FALSE
```

### 9.1.5 String definition

String definitions start and end with a double quote: `"`. If you want to insert double quotes within a string use either `\"` or `\q`. Double quotes in strings are written as `\q` in the output file. This ensures that no software package that uses the file will be confused.

### 9.1.6 Mathematical operators

| + | addition, string concatenation |
|---|---|
| − | subtraction |
| * | multiplication |
| / | division |

### 9.1.7 Relational operators

| < | smaller |
|---|---|
| <= | smaller or equal |
| == | equals |
| != | unequal |
| >= | greater or equal |
| > | greater |

### 9.1.8 Logical operators

| & | logical and |
|---|---|
| \| | logical or |

### 9.1.9 Scope operators

| : | next higher scope |
|---|---|
| \ | highest possible scope (this is always the record of the globals table) |

### 9.1.10 Statements

In the following, $a$ and $b$ are logical expressions, $x$, $y$, and $z$ numeric expressions, $v$ is the name of a variable, $t$ is the name of a table, $s$ is one statement, $ss$ and $sse$ are lists of statements.

| | |
|---|---|
| `x = y;` | Assigns the expression $y$ to the variable $x$. |
| `if ( a ) { ss }` | If $a$ returns TRUE, the statements $ss$ are executed. |
| `if ( a ) { ss₁ } else {ss₂}` | If $a$ returns TRUE, the statements $ss_1$ are executed; otherwise the statements $ss_2$ are executed. |
| `if (a ) { ss₀ }` `elsif ( b₁ ) { ss₁ }` `elsif ( b₂ ) { ss₂ } …` | If $a$ returns TRUE, the statements $ss_0$ are executed; otherwise, if $b_1$ returns TRUE, the statements $ss_2$ are executed and so on. |
| `if ( a ) { ss₀ }` `elsif ( b₁ ) { ss₁ }` `elsif ( b₂ ) { ss₂ } …` | If $a$ returns TRUE, the statements $ss_0$ are executed; otherwise, if $b_1$ returns TRUE, the statements $ss_1$ are executed and so |

| | |
|---|---|
| `else { sse }` | on. If none of either the `a` or `b_i` statements are TRUE, the statements `sse` are executed. |
| `t.do { ss }` | In the table `t`, the statements `ss` are executed for all records. |
| `t.new { ss }` | In the table `t`, a new record is generated and in this record the statements `ss` are executed. |
| `array v[ x ];` | Defines an array, i.e., an indexed variable. Array variables can be accessed with `v[ x ]`. Index runs from 1 to `x` in steps of 1. |
| `array v[ x, y ];` | Defines an array, i.e., an indexed variable. Array variable can be accessed with `v[ x ]`. Index runs from `x` to `y` in steps of 1. |
| `array v[ x, y, z ];` | Defines an array, i.e., an indexed variable. Array variable can be accessed with `v[ x ]`. Index runs from `x` to `y` in steps of `z`. |
| `while( a ) { ss }` | While `a` is TRUE, the statements `ss` are executed. Loops can be left with the key combination **Ctrl+Alt+F5**. |
| `repeat { ss } while ( a );` | The statements `ss` are executed. Then it is checked whether `a` is TRUE. As long as `a` is TRUE `ss` is repeated. Loops can be left with the key combination **Ctrl+Alt+F5**. |
| `for ( v ) { ss }` | The variable `v` is set sequentially to 1, 2, … number of subjects. For each of these value the statements `ss` are executed. |
| `for ( v, n ) { ss }` | The variable `v` is set sequentially to 1, 2, … `n`. For each of these value the statements `ss` are executed. |
| `for ( v, n_1, n_2 ) { ss }` | The variable `v` is set sequentially to $n_1$, $n_1$+1, … $n_2$. For each of these value the statements `ss` are executed. |
| `for ( v, n_1, n_2 , step ) { ss }` | The variable `v` is set sequentially to $n_1$, $n_1$+$step$, … $n_2$. For each of these value the statements `ss` are executed. The value of `step` can be negative. |
| `later ( a ) do { ss }` | The expression `a` is calculated. The resulting seconds later the statements `ss` are executed. The statements `ss` are executed in the exact same scope environment in which this later command was called. If `a` is negative, `ss` is not called. |
| `later ( a ) repeat { ss }` | The expression `a` is calculated. The resulting seconds later the statements `ss` are executed. The statements `ss` are executed in the exact same scope environment in which this later command was called. After this, condition `a` is calculated again and `a` seconds later `ss` is executed again. If `a` is negative the later command is cancelled. At the end of the period in which this later command was started, it is also cancelled. |

### 9.1.11 Functions

In the following, $a$ and $b$ are conditions, i.e. boolean expressions.

| | |
|---|---|
| abs( $x$ ) | Absolute value of $x$. |
| and( $a$, $b$ ) | TRUE if and only if $a$ and $b$ are true. |
| atan( $x$ ) | Arctangent of $x$. Result is in radians. |
| cos( $x$ ) | Cosine function of $x$. Argument is in radians. |
| exp( $x$ ) | Exponential of $x$; $e^x$. |
| gettime() | The number of seconds since the computer was started. |
| if( $a$, $x$, $y$ ) | If $a$, then the value of the function is $x$, otherwise $y$. |
| ln( $x$ ) | Natural logarithm of $x$. |
| log( $x$ ) | Base -10 logarithm of $x$. |
| max( $x$, $y$ ) | Maximum of $x$ and $y$. |
| min( $x$, $y$ ) | Minimum of $x$ and $y$. |
| mod( $x$, $y$ ) | Remainder after $x$ is divided by $y$. |
| not( $a$ ) | TRUE if and only if $a$ is not true. |
| or( $a$, $b$ ) | TRUE if and only if $a$ or $b$ is true. |
| optionswitch( $v$, $opt1$, $res1$, $opt2$, $res2$, ... ) | Corresponds to the logic of the !text-layout: find the first best correspondence of $v$ within the options $opt\_i$ and return $res\_i$. |
| pi() | 3.1415… |
| power( $x$, $y$ ) | $x^y$, if $x$ is positive. If $x$ is negative and $y$ is an odd number or $1/y$ is an odd number, $x^y$ is returned, otherwise $|x|^y$ is returned. |
| random() | Uniformly distributed random number between 0 and 1. |
| randomgauss() | Normally distributed random number with average 0 and standard deviation 1. |
| randompoisson( $x$ ) | Poisson distributed random number with average $x$ (the result is a whole number). |
| round( $x$, $y$ ) | Rounds $x$ to a multiple of $y$, i.e., supplies the multiple of $y$ that is closest to $x$. |
| rounddown( $x$, $y$ ) | Rounds $x$ down to a multiple of $y$, i.e., it returns the greatest multiple of $y$ that is smaller or equal to $x$. This definition is also employed for negative numbers. |
| roundup( $x$, $y$ ) | Rounds $x$ up to a multiple of $y$, i.e., it returns the smallest multiple of $y$ that is greater than or equal to $x$. This definition is also employed for negative numbers. |
| same( $x$ ) | Short form for $x$ == :$x$. Therefore, this function is only feasible in table functions. $x$ may also be an expression. |
| sin( $x$ ) | Sine of $x$. Argument is in radians. |

```
sqrt( x )                    Square root of x.
```

**Examples of optionswitch**

```
s = optionswitch( 1, 0, "zero", 1, "one", 2, "two" );  // "one"
c = optionswitch( TRUE, FALSE, red(), TRUE, blue()); // rgb(0,1,0)
```

**Rounding examples**

```
x = round( 345.67, 20 );   // 340
x = round( 345.67, 0.1 );  // 345.7
y = round( 2.5, 1 );       // 3
y = round( -2.5, 1 );      // -3;
z = rounddown( -3.6, 1 );  // -4;
z = roundup( -3.6, 1 );    // -3;
```

### 9.1.12 String functions

In the following, $x$, $y$, and $z$ are numeric expressions, and $s$, $t$, and $u$ are string expressions.

> Some characters will be replaced in the output file: the quote is replaced by \q, new line by \n, tab by \t, and backslash by \\.

| | |
|---|---|
| `char( x )` | Returns the character specified by a numeric code $x$ of the used character set. $x$ is a number between 1 and 255. |
| `code( s )` | Returns a numeric code for the first character in string $s$. The returned code depends on the character set used. |
| `mid( s, x, y )` | Returns the middle of the string $s$, starting at character number $x$ and going for $y$ characters. The counting process starts at 1, that is the first character has a place value of 1. |
| `pos( s, t, x )` | Returns place number where string $t$ occurs in string $s$, starting the counting process at number $x$. |
| `len( u )` | Returns the number of characters in string $u$. |
| `upper( s )` | Returns string $s$ in capital letters. |
| `lower( s )` | Returns string $s$ in lower case letters. |
| `trim( s )` | Trims the outside spaces around text letters, but not spaces within text. |
| `format( x, y )` | Returns $x$ as a string variable rounded to the precision of $y$. |
| `stringinsert( s, x, s1 )` | Returns $s$, in which the string $s1$ is introduced at position $x$. |
| `stringdelete( s, x, y )` | Returns $s$, in which at position $x$ $y$ characters are deleted. |
| `replaceall( s, s1, s2 )` | Returns $s$, in which all occurrences of $s1$ are replaced by $s2$. |

| | |
|---|---|
| `stringtonumber( s )` | Converts string *s* to a number. If the string does not represent a number, 0 is returned. |
| `stringtonumber( s, x )` | Converts string *s* to a number. If the string does not represent a number, *x* is returned. |
| `countstringsubstrings( s, substring )` | Returns the number of occurrences of *substrings* in *s*. |
| `countstringsubstringwithnonseparators( s, substring, s_letters )` | Returns the number of occurrences of *substrings* in *s*. Acceptable are only substring that are not neighboring a letter in *s_letters*. The string with the letters can contains groups of letters. They are specified by repeating the first character of the group as in "aaz". |
| `countstringsubwords( s, substring, s_letters )` | Returns the number of occurrences of *substrings* in *s*. Acceptable are only word, i.e. string that are surrounded with non-alphanumeric characters. The string with the letters can contains groups of letters. They are specified by repeating the first character of the group as in "aaz". |
| `countstringsubstringwithseparators( s, substring, s_letters )` | Returns the number of occurrences of *substrings* in *s*. Acceptable are only substring that are not neighboring the separators specified in *s_letters* The string with the letters can contains groups of letters. They are specified by repeating the first character of the group as in "aaz". |
| `countstringletters( s, s_letters )` | Returns the number of occurrences of any letter in *s_letters* within the string *s*. The string with the letters can contains groups of letters. They are specified by repeating the first character of the group as in "aaz". |

**String function examples**

```
s = "World";
t = "Hello " + s; // "Hello World"
c = char(65);   // "A" for the ASCII character set
x = code("A");  // 65 for the ASCII character set
p1 = pos("find in this string", "in", 0);     // 2, found in "find"
p2 = pos("find in this string", "in", p1 + 1); // 6, found in "in"
p3 = pos("find in this string", "na", 0);     // 0, "na" not found
l = len(trim("  foo   ")); // 3
f = format(12.34, 0.1); // "12.3"
z = stringtonumber(f); // 12.3
n1 = countstringsubstrings ( "abcdbcf bc f", "bc"); // 3
n2 = countstringsubstringwithnonseparators( "bcxybc-xbc", "bc", "x"); // 1
n3 = countstringsubwords("this, and that should count as this", "this"); // 2
n4 = countstringsubstringwithseparators( "bcxybc xbc-xbc", "bc", "x"); // 2
n5 = countstringletters( "aaaabbbcccdefgh", "bbcf"); // 7
```

### 9.1.13 Color functions

In the following specifications, we always mention the type of the variables.

| | |
|---|---|
| rgb( *r*, *g*, *b* ) | Returns the color with red, greed and blue intensities r, g, and b. Types: r, g, b: reals between 0 and 1; result: color. |
| rgb255( *r*, *g*, *b* ) | Returns the color with red, greed and blue intensities r, g, and b. Types: r, g, b: reals between 0 and 255; result: color. |
| hsl( *h*, *s*, *l* ) | Returns the color with hue, saturation and lightness h, s, and l. Types: h, s, l: reals between 0 and 1; result: color. |
| hsl255( *h*, *s*, *l* ) | Returns the color with hue, saturation and lightness h, s, and l. Types: h, s, l: reals between 0 and 255; result: color. |
| black() | Returns the color black == rgb(0,0,0) == hsl(0,0,0). |
| white() | Returns the color white == rgb(1,1,1) == hsl(0,0,1). |
| red() | Returns the color red == rgb(1,0,0) == hsl(0,1,0.5). |
| green() | Returns the color green == rgb(0,1,0) == hsl(1/3,1,0.5). |
| blue() | Returns the color blue == rgb(0,0,1) == hsl(2/3,1,0,5). |
| yellow() | Returns the color yellow == rgb(1,1,0) == hsl(1/6,1,0,5). |
| cyan() | Returns the color cyan == rgb(0,1,1) == hsl(3/6,1,0,5). |
| magenta() | Returns the color magenta == rgb(1,0,1) == hsl(5/6,1,0,5). |
| getred(*c*), getred255(*c*) | Returns the red component of the color c as a number between 0 and 1, or 0 an 255 respectively. |
| getgreen(*c*), getgreen255(*c*) | Returns the green component of the color c as a number between 0 and 1, or 0 an 255 respectively. |
| getblue(*c*), getblue255(*c*) | Returns the blue component of the color c as a number between 0 and 1, or 0 an 255 respectively. |
| gethue(*c*), gethue255(*c*) | Returns the hue component of the color c as a number between 0 and 1, or 0 an 255 respectively. |
| getsaturation(*c*), getsaturation255(*c*) | Returns the saturation component of the color c as a number between 0 and 1, or 0 an 255 respectively. |
| getlightness(*c*), getlightness255(*c*) | Returns the lightness component of the color c as a number between 0 and 1, or 0 an 255 respectively. |
| getint(*c*) | Returns an ID for the color. for rgb255(r,g,b) it equals $r + 256*g + 256^2*b$. |
| mixlinear(*c1*, *c2*, *w*) | Returns a color which is a component wise linear combination of colors c1 and c2 where c1 is weighted with w. Each component is rounded up |
| mixlinear(*c1*, *c2*) | Returns a color which is a component average of colors c1 and c2. |
| mix(*c1*, *c2*, *w*) | Mixes the colors more naturally than mixlinear because it applies a quadratic transformation to each component. For |

|  |  |
|---|---|
|  | example, mix(white(), black(),0.5) returns a gray with a value of sqrt(0.5) |
| `mix(c1, c2)` | Returns mix(c1, c2, 0.5)`. |
| `colorcomplexity(c)` | Returns how many steps are at least necessary in order to create this color by using mixlinear starting with basic colors. |
|  | (colorcomplexity |

**Color function examples**

```
c = rgb(1,0,.5);
c2 = rgb255(255,128,0);
c3 = mix(c1,c2); // rgb255(255, 91, 91)
r = getred(c3); // 1
v = sqrt(0.5);
mediumgray = rgb(v,v,v);
```

### 9.1.14 Table functions

| | |
|---|---|
| `average( x )`, `average( a, x )` | Average of the (found) numeric values. |
| `count()`, `count( a )` | Number of records in the table or number of found records. |
| `find( x )`, `find( a, x )`, `find( s )`, `find( a, s )` | The first value of the variable (where $a$ is satisfied). |
| `maximum( x )`, `maximum( a, x )` | Maximum of the (found) numeric values. |
| `median( x )`, `median( a, x )` | Median of the (found) numeric values. |
| `minimum( x )`, `minimum( a, x )` | Minimum of the (found) numeric values. |
| `product( x )`, `product( a, x )` | Product of the (found) numeric values. |
| `regressionslope( x, y )`, `regressionslope( a, x, y )` | Gradient of a linear regression through the (found) points $(x, y)$. |
| `stddev( x )`, `stddev( a, x )` | Standard deviation of the (found) numeric values. |
| `sum( x )`, `sum( a, x )` | Sum of the (found) numeric values. |

### 9.1.15 Iterator

In order to simplify the programming language and enabling speed improvements, the iterator is not supported in the versions later than 3.7.0. Use the `for` loop instead. There are three changes to make:

1. Replace `iterator` by `for`

2. Remove `.do`

3.  Change the scope in order to reflect the fact that the `for` loop does not create an new scope environment.

Example:

```
// old iteration with iterator
S=0;
iterator(i, 5).do{
  :S = :S + i;
}

// new iteration with for
S=0;
for(i, 5) {
  S = S + i;
}
```

See also Section 3.10.2, "Loops: while, repeat and for" in the tutorial.

### 9.1.16 Syntax diagram of the programming language

The following syntax description is intended to serve as a reference to experts.

**Syntax used to express the grammar**

`a : b`    a can be replaced by `b`

`a b`      a followed by `b`

`a | b`    a or `b`

`"x"`      exactly that text

`a*`       zero, one or more repetitions of `a`

`(a)`      a

`'x'`      the character `x`

## Definitions

```
digit19 : '1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9';


digit09 : '0'| digit19;


letter : '_'|
         'a'|'b'|'c'|'d'|'e'|'f'|'g'|'h'|'i'|'j'|'k'|'l'|'m'|
         'n'|'o'|'p'|'q'|'r'|'s'|'t'|'u'|'v'|'w'|'x'|'y'|'z'|
         'A'|'B'|'C'|'D'|'E'|'F'|'G'|'H'|'I'|'J'|'K'|'L'|'M'|
         'N'|'O'|'P'|'Q'|'R'|'S'|'T'|'U'|'V'|'W'|'X'|'Y'|'Z';


integer : ( digit19 digit09* ) | '0';


realdigit : integer '.' digit09 * | '.' digit09 digit09* ;


real : realdigits ( ('e' ('+'|'-'|"") integer) | "");
```

## Terminal elements

```
NAME : letter (letter|digit)*;


NUMBER : real;


BOOLVALUE : "FALSE" | "TRUE";


FUNCTIONNAME0 : "gettime" | "pi" | "random" | "randomgauss" ;


FUNCTIONNAME1 : "abs" | "atan" | "char" | "code" | "cos" | "exp"
                | "len" | "ln"| "log"| "randompoisson" | "sin"
                | "sqrt" | "trim" | "upper" | "lower";


FUNCTIONNAME2 : "format" | "max" | "min" | "mod" | "power"
                | "round" | "rounddown" | "roundup" | "stringtonumber";


FUNCTIONNAME3 : "mid" | "pos";


BOOLFUNCTIONNAME1 : "not";


BOOLFUNCTIONNAME2 : "and" | "or";


BOOLFUNCTIONNAME1R : "same";


TABLEFUNCTION0 : "count";


TABLEFUNCTION1 : "average" | "find" | "maximum" | "median"
                | "minimum"| "product" | "stddev" | "sum";


TABLEFUNCTION2 : "pearsoncorr" | "regressionslope" | "spearmancorr";


BOOLOPERATOR2 : "&" | "|";


COMPOPERATOR : "<" | "<=" | "==" | "!=" | ">=" | ">" ;


SCOPEOPERATOR : ":" | "\";
```

# Grammar

```
program : statementlist;


statementlist :
                | statement statementlist
                ;


statement : ';'
            | lvalue '=' expression ';'
            | "while" '(' boolexpression ')' '{' statementlist '}'
            | "repeat" '{' statementlist '}' "while" '(' boolexpression ')' ';'
            | "later" '(' expression ')' "do" '{' statementlist '}'
            | "later" '(' expression ')' "repeat" '{' statementlist '}'
            | "do" '{' statementlist '}'
            | tableidentifier '.' "do" '{' statementlist '}'
            | "new" '{' statementlist '}'
            | tableidentifier '.' "new" '{' statementlist '}'
            | "if" '(' boolexpression ')' '{' statementlist '}' elseclause
            | "array" arraydeclaration ';'
            ;


elseclause :
            | "elsif" '(' boolexpression ')' '{' statementlist '}' elseclause
            | "else" '{' statementlist '}'
            ;


lvalue : variable;


arraydeclaration : purearraydeclaration
                 | typeidentifier purearraydeclaration
                 ;


typeidentifier : "real" | "string" | "boolean" | "color"


purearraydeclaration  : IDENTIFIER '[' ']'
                      | IDENTIFIER '[' expression ']'
                      | IDENTIFIER '[' expression ',' expression ']'
                      | IDENTIFIER '[' expression ',' expression ',' expression ']'
                      ;


expression : summand
            | '+' expression
            | '-' expression
            | expression '+' expression
            | expression '-' expression
            ;
```

```
summand : factor
        | summand '*' summand
        | summand '/' summand
        ;
```

```
factor : STRING
       | NUMBER
       | variable
       | '(' expression ')'
       | FUNCTIONNAME0 '(' ')'
       | FUNCTIONNAME1 '(' expression ')'
       | FUNCTIONNAME2 '(' expression ',' expression ')'
       | FUNCTIONNAME3 '(' expression ',' expression ',' expression ')'
       | "if" '(' boolexpression ',' expression ',' expression ')'
       | tablefunction
       ;

variable : IDENTIFIER
         | arrayvariable
         | SCOPEOPERATOR variable
         ;

arrayvariable : IDENTIFIER '[' expression ']'
              ;

tablefunction : puretablefunction
              | tableidentifier '.' puretablefunction
              ;

puretablefunction : TABLEFUNCTION0 '(' ')'
                  | TABLEFUNCTION0 '(' boolexpression ')'
                  | TABLEFUNCTION1 '(' expression ')'
                  | TABLEFUNCTION1 '(' boolexpression ',' expression ')'
                  | TABLEFUNCTION2 '(' expression ',' expression ')'
                  | TABLEFUNCTION2 '(' boolexpression ',' expression //
                                                ',' expression ')'
                  ;

tableidentifier : IDENTIFIER
                | ITERATOR '(' IDENTIFIER ')'
                | ITERATOR '(' IDENTIFIER ',' expression ')'
                | ITERATOR '(' IDENTIFIER ',' expression ',' expression ')'
                | ITERATOR '(' IDENTIFIER ',' expression ',' expression //
                                                ',' expression ')'
                ;
```

```
boolexpression : boolatom
               | boolatom OPBOOL boolexpression
               ;


boolatom : BOOLFUNCTIONNAME0
         | expression OPCMP expression
         | '(' boolexpression ')'
         | BOOLFUNCTIONNAME1 '(' boolexpression ')'
         | BOOLFUNCTIONNAME2 '(' boolexpression ',' boolexpression ')'
         | BOOLFUNCTIONNAME1R '(' expression ')'
         ;
```

## 9.2 Scope environment

When a program is executed, it is always executed within a particular scope environment. In the following, we give a description of what environment the programs are called in.

### 9.2.1 Program at the beginning of a stage

The following table lists the scope environment for programs that are run at the beginning of a stage.

| Table | Owner Variable | Scope environment | Records for which the program is run |
|---|---|---|---|
| globals | not allowed | globals | The one record of the globals table. If there is a condition, the program is only executed if the condition is satisfied. |
| summary | | summary : globals | Record of current period. If there is a condition, the program is only executed if the condition is satisfied. Remark: To run a program in all records of the summary table, use `summary.do` in a program. |
| subjects | | subjects : globals | All records in the subjects table. If the stage does not wait for all then whenever the subject enters the program is executed for the corresponding subject. |
| session | | session : subjects : globals | All records in the session table. The scope operator allows accessing the corresponding record of the subjects table. If there is a condition, the program is only executed if the condition is satisfied. |
| All other tables | without | table: globals | All records in the table. If there is a condition, the program is only executed if the condition is satisfied. |
| All other tables | with | table : subjects : globals | All records in the table. If the owner variable is `Owner` then the scope operator accesses the record in the subjects table for which `Subject` equals `Owner`. This corresponds to a program in the subjects table: `table.do{ if (Owner == :Subject){ … } }`. If there is a condition, the program is only executed if the condition is satisfied. |

### 9.2.2 Program in a button or in a plot item

The button or the plot input item sets a scope environment. It includes the globals table and the record of the subject who initiated the action, e.g. presses the button. Then it includes the record that is selected or created of the chain of record in the case of a plot input item in nested plot graphs. This is the basic scope. We call *target table* the table at the beginning of this scope chain. This is for example the table in which a record is selected in a contract list box.

A program in the target table *T* is executed in this scope environment if there is no condition. For all other tables *X* or if there is a condition in the program of the target table, then the program is executed

in this table *X* for all records (satisfying the condition). In this case, the scope environment is the current record of the target table and the basic scope, i.e. subject and globals table.

**Examples**

There is a contract list box displaying record in the contracts table. A program in the contracts table without condition has the scope environment contracts : subject : globals. It is executed in the relevant record of the contracts table. A program in the contracts table with condition has also the scope environment contracts : subject : globals but it is executed for all records satisfying the condition. A program in another table has the scope environment table : subject : globals.

If you want to use the variable of the relevant record of the target table *T* in another table *X* then use a program in the target table and execute the program in the other table using the do command.

Owner variables are not allowed for programs in buttons and plot input items.

**9.2.3 Special case contract creation box**

The new record is not yet included in the database. Thus programs in the contracts table with a condition cover only the old records.

**9.2.4 Special case multiple record creation**

In the contract creation box more than one record can be created at a time. Similarly, in the contract grid box, a program refers to more than one record. In this case programs in the target table without a condition are executed for this record. Programs in other tables *X* and programs in the target table with a condition are executed for all the record in *X* as well as the selected records in the target table. The following rules apply. The programs are run one after the other for all necessary record combinations. Within a program, the program runs through the records of *X* for the first relevant record in the target table, then through the records of *X* for the second relevant record in the target table and so on.

**9.2.5 Summary**

In a nutshell, programs executed at the beginning of the period are executed in the corresponding table with access to the globals table and other tables in the scope environment. Programs in buttons and plot input items are executed in a scope environment that allows to access the record of the subject who pressed the button and the structure selected.

**9.3 Program execution**

Programs at the beginning of a stage are executed one after the other. If the Start option in the stage is set to Wait for all, the procedure is as follows: First, the first program is executed for all subjects. Then the next program is executed for all subjects and so forth. Note that setting the value of `Participate` to zero does not stop program execution. `Participate` does not take effect until all programs have been executed.

If another Start option is set, first, the first program is executed for the subject who entered first. Then the second program is executed for this subject and so far. When all programs for the first subject are executed, then the programs for the next subjects are executed. In this case not all programs are executed for all subjects. The programs in the globals table are only executed for the first subject, programs for the summary table only for the last one. Programs in the subjects and session table are executed for the record of the subject. For user-defined tables, it is specified in the table definition whether the program is executed at the beginning, at the end, or for every subject. If there is an owner variable in a program of the contracts table (or a user defined table), the program is only executed for the records of the owner. This means that it is executed for the records in which the owner variable equals the value of the given `Subject` variable.

At the beginning of each period, the programs are executed in the following order:

1.  Standard variables are set

2.  Programs in background

3.  Subject program (in current period) in the subjects table (the program in the specific parameter)

4.  Role program in subjects table (in the role parameter)

5.  Period program in globals table (in the period parameter)

6.  Programs of the first stage

### 9.4 The tables and their standard variables

### 9.4.1 The subjects table

The subjects table contains a record for each subject. This is the main table. It is freshly set up, i.e. created anew, each period. The subjects table of the preceding period is available under the name OLDsubjects. Values from earlier periods than the immediately preceding one are not accessible.

In the subjects table, the variables `Period`, `Subject`, `Group`, `Profit`, `TotalProfit` and `Participate` are always defined.

`Period`

> Number of the period. The first paying period is period 1. Thus, if there are 3 trial periods, the first period has the number -2.

`Subject`

> Number of subject; starts at 1.

`Group`

> Group number as it is displayed in the parameter table (possibly modified by a program).

`Profit`

Profit made in this period (in experimental currency units). Needs to be calculated; default is zero. `Profit` is the relevant variable for payment.

`TotalProfit`

Total profit made in this treatment. This is calculated automatically. It should not be changed.

`Participate`

Indicates whether the subject is taking part in the current stage or not. If `Participate` equals 1, he or she is taking part, if 0, he or she is not taking part. Default is 1.

`LeaveStage`

If set to 1, it causes a subject to leave an active state and to move to the waiting state.

### 9.4.2 The globals table

This table contains a single record, i.e. a single row. In it, values are stored that are the same for all subjects, as for instance are global variables. It is freshly set up, i.e. created anew, for each period. The globals table of the preceding period is available under the name OLDglobals. It contains the following variables per default:

`Period`

As in subjects table.

`NumPeriods`

ID of the last period.

`RepeatTreatment`

If this variable is greater than zero after the last period, the treatment is run again with the same number of periods. The period counter is incremented as if it is all done in one treatment.

### 9.4.3 The summary table

The summary table contains one record per period. This record is not destroyed when the period has been finished. The whole table is destroyed after the treatment has been finished. This table is most useful for storing and observing aggregate data of the treatment.

`Period`

Number of periods as in the subjects table.

### 9.4.4 The session table

As in the subjects table, this table contains one record per subject. However, this table is not freshly set up after each period. It contains the aggregated profits earned in earlier treatments. You can also use it to exchange information between treatments.

Subject

> Subject ID.

FinalProfit

> Income of the subject not including the show up fee.

ShowUpFee

> Show up fee.

ShowUpFeeInvested

> 1, if subject agreed to invest show up fee to be able to proceed in the experiment.

MoneyAdded

> Credit limit of the subject.

MoneyToPay

> FinalProfit + ShowUpFee + MoneyAdded.

MoneyEarned

> FinalProfit + ShowUpFee.

### 9.4.5 The contracts table

This table is used mainly for market experiments. New records can be added to this table and existing records can be changed. Section 3.6, "Continuous Auction Markets" explains the procedure in detail.

Period

> Period in which the record was created.

### 9.4.6 The table of the previous period "OLDtables"

The data of past periods can be accessed in the OLD-tables. You just have to prefix the table name with OLD. Data of even earlier periods have to be accessed indirectly or by using a table with lifetime treatment or session.

### 9.4.7 User defined tables

User defined tables can be declared at the beginning of the treatment.

`Period`

    Period in which the record was created.

# 10 Text Formatting

## 10.1 Variable output in text display

Besides putting a variable into an item, there is a second way to display the value of a variable. It can also be inserted in text. The syntax is

```
< variable | layout >
```

In the output, this expression is replaced by the value of the `variable` in the given `layout`. Layout can be a *layout* command as it is defined in the Layout field of the item. Of course, only the number and the `!text` options are valid. For instance, you cannot insert buttons in this way.

Variables can be inserted in labels and in the options of a layout of an item or in the text of a help box. So it is for instance possible that the text of a radio button depends on what happened before in the experiment. If a text contains inserted variables it must be preceded by `<>`, otherwise the text will not be processed. The characters `<>` have to appear at the very beginning of the dialog field. It is also possible to insert variables in the options of the layout of an inserted variable, i.e., to nest inserted variables. If a variable appears in an option of the same variable, the variable name can be omitted. So

```
<x|!text: -1="negative"; 1="<|1>";>
```

is a shorter form for

```
<x|!text: -1="negative"; 1="<x|1>";>
```

> ⚠️ Be careful when you use this option for items that can change their value. The width of an item is calculated when the screen is displayed first and it is not modified afterwards. So, if the value of a variable changes from 1 to 20, only 2 might be displayed. You are responsible for avoiding this. You can choose a sufficiently wide first value or place the item into a box with other items that are wider. You could use an invisible variable that contains a lot of spaces.

Note further that labels are evaluated only once, at the beginning of the stage. Variables in labels are not updated.

## 10.2 Formatting with RTF commands known to be processed

In some places, texts can be formatted with RTF. Labels can be formatted in standard boxes, grid boxes, contract creation boxes and contract grid boxes. The options in the `!text` layout can be formatted in standard boxes, grid boxes, contract creation boxes and contract grid boxes. Finally, texts in help boxes and message boxes can be formatted.

The RTF format begins with "{\rtf " (with a blank space at the end) and ends with "}". In between is the text to which formatting instructions can be added. In options of `!text` layouts, the "{\rtf " has to appear in every option:

```
!text: 1="{\rtf \b one}"; 2="{\rtf \b two};
```

Formatting instructions begin with \ and end with a blank space. If a formatting option is supposed to apply only to a certain range, you can place this range in curly brackets. Not the whole RTF is supported. The most important formatting instructions that are supported are:

| | |
|---|---|
| `\tab` | Tabulator |
| `\par` | New paragraph |
| `\line` | New line |
| `\bullet` | Bullet |
| `\ql` | Aligned to left |
| `\qr` | Aligned to right |
| `\qc` | Centered |
| `\b` | Bold |
| `\b0` | Not bold |
| `\i` | Italic |
| `\i0` | Not italic |
| `\sub` | Small and inferior numbers (index) |
| `\super` | Small and superior numbers (exponent) |
| `\strike` | Crossed through |
| `\ul` | Underline |
| `\ul0` | Do not underline |
| `\colortbl` | Color table. See examples. |
| `\cf`*n* | Text color. *n* is the index of the color table which is defined by `\colortbl`. |
| `\fs`*n* | Font size *n* in units of half a dot. The font size must be explicitly given, otherwise it is larger (24) than usual in z-Leaf. |
| `\ansicpg`*n* | ANSI code page. *n* represents the code page in decimal. See Section 11.1, "Command line options common for z-Tree and z-Leaf" for a list of the supported languages and the corresponding code pages. |

For more complex operations it is best to format the text in a word processor and then export it as RTF. However, if you create the RTF code by hand, it will be shorter and easier to read.

**Examples**

```
{\rtf \fs18 normal font size, \b bold, \b0 no longer bold}
```

normal font size, **bold**, no longer bold

```
{\rtf \fs18 Text {\i italic} no longer italic \par new line}
```

Text *italic* no longer italic
new line

```
{\rtf {\colortbl;\red0\green0\blue0;\red128\green128\blue0;}
\fs18 One word in \cf2 olive\cf1 , the rest in black.}
```

One word in olive, the rest in black.

## 10.3 Combining RTF and inserted variables

The insertion of variables is carried out *before* the interpretation of RTF. This makes conditional formatting possible as in the following example. When the variable BOLD is 1, "hallo" should be shown in boldface but otherwise in plain text.

```
<>{\rtf <BOLD|!text:0="";1="\b ";>hallo}
```

# 11 Command Line Options

## 11.1 Command line options common for z-Tree and z-Leaf

`/language` *lan*

This option sets the language as in the menu command Treatment → Language. In z-Tree it concerns the default element of a treatment and questionnaire. In z-Leaf it is relevant for the general error messages. The default value is German. If you want to change this in your environment, you can create a shortcut for z-Tree that initializes the language of your choice.

In the following list, we give all the options available. For instance, to use Finnish, you can use the option `/language suomi`. Note that for the Russian and for the Ukraine version, you need a Russian or Ukraine operating system.

| Language | Command Line Option | Codepage |
| --- | --- | --- |
| Arabic | `ar, arab, arabic` | 1256 |
| Bahasa Indonesia | `id, indonesian` | 1252 |
| Brasil | `br, brasil` | 1252 |
| Catalan | `cat, catala, catalan` | 1252 |
| Chinese | `cn, chinese` | 936 Simplified GBK |
| Croatian | `hr, hrvatski, croatian` | 1250 |
| Czech | `cz, cesky, czech` | 1250 |
| Danish | `dk, dansk, danish` | 1252 |
| Dutch | `nl, nederlands, dutch` | 1252 |
| English | `en, english` | 1252 |
| Finnish | `fi, suomi, finnish` | 1252 |
| French | `fr, francais, french` | 1252 |
| German | `de, deutsch, german` | 1252 |
| Greek | `gr, greek` | 1253 |
| Hebrew | `il, ivrit, hebrew` | 1255 |
| Italian | `it, italiano, italian` | 1252 |
| Japanese | `jp, nihongo, japanese` | 932 Shift-JIS |
| Magyar | `hu, magyar, hungarian` | 1250 |
| Melayu | `ms, melayu, malay` | 1252 |
| Mongolian | `mng, mongolian` | 1251 |
| Norwegian - Bokmål | `no, norsk, norwegian` | 1252 |
| Norwegian - Nynorsk | `nyno, nynorsk, newnorwegian` | 1252 |
| Polish | `pl, polski, polish` | 1250 |
| Portugues | `pt, portuguese, portugues` | 1252 |

| Language | Command Line Option | Codepage |
|----------|---------------------|----------|
| Russian | `ru, ruskii, russian` | 1251 |
| Slovenian | `sv, slovensky, slovak` | 1250 |
| Spanish | `es, espanol, spanish` | 1252 |
| Swedish | `se, svensks, swedish` | 1252 |
| Swiss German Zurich style | `zh, zuerituetsch, zurichgerman` | 1252 |
| Turkish | `tr, turkce, turkish` | 1254 |
| Ukraine | `ua, ukrainska, ukraine` | 1251 |
| Vietnamese | `vi, vietnamese` | 1258 |

`/channel ch`

Determines the channel through which the z-Leaves establish contact to z-Tree. If you want to run more than one z-Tree on one computer they must work with different channels, i.e., you set the channel to values greater than one. Defaults to 0.

## 11.2 Command line options for z-Tree

`/xlsdir dir`

Sets the directory where the xls file is stored.

`/sbjdir textsldir`

Sets the directory where the subjects file is stored.

`/zttdir dir`

Sets the directory where the treatment and questionnaire files are automatically stored (before they run).

`/datadir dir`

Sets the directory where the xls and the subjects file (sbj) are stored as well as treatment and questionnaire files.

`/adrdir dir`

Sets the directory where the address file is stored.

`/paydir dir`

Sets the directory where the payment file is stored.

`/privdir dir`

Sets the directory where the address and the payment file are stored.

`/gsfdir` *`dir`*

Sets the directory where the *GameSafe* is stored.

`/tempdir` *`dir`*

Sets the directory where the temporary files `@lastclt.txt`, `@db.txt`, and `@prevdb.txt` are stored.

`/leafdir` *`dir`*

Sets the directory where the file `server.eec` is stored.

`/treatment` *`file`*

Opens the treatment specified in *`file`* and starts it as soon as sufficiently many clients are connected.

`/checkdiskspace` *`size`*

Checks whether there is sufficient disk space on the hard drive.

`/asksessionname`

When z-Tree starts, a prompt appears in which a name for the session can be entered. The name is used for file names and for the session name in the output. It is enforced that the name contains current date and time.

`/sessionprefix` *`prefix`*

Change the session name.

## 11.3 Command line options for z-Leaf

`/server` *`adr`*

*`adr`* contains the IP address of the experimenter PC.

`/name` *`name`*

*`name`* is the name the client has in the clients' table.

`/nameisipaddress`

The IP address is the name the client has in the clients' table.

`/size` *`widthxheight`*

Size of the client's window. In this way you can test on a large screen how the layout will look on a small screen. For example, the VGA format is `640x480`. There may be no spaces around the 'x'.

`/position xpos,ypos`

Position of the client's window with respect to the top left corner. There may be no spaces around the comma.

`/fontsize size`

Sets the font size.

`/fontface face`

Sets the font face.

`/buttonfontface face`

Sets the font face for buttons.

`/logtime`

Stores time at which input is made or the screen is updated. It uses the time on z-Leaf. This time is therefore not influenced by network delays. On the other hand, this time is not synchronized across different subjects.

`/logtimelocal`

The timing information is stored on a local directory on the z-Leaf computer.

`/xlsdir dir`

Location where the time file is stored locally, on the z-Leaf computer.

`/xlsdirsrv dir`

Location where the time file is copied when the experiment is finished. It prevents network traffic during the experiment.

`/counttriggers control_register,bit_to_test`

Instructs z-Leaf to read the parallel port and record triggers from external hardware (i.e. fMRI scanner).

Example: `/counttriggers 957,5=1` tells z-Leaf to check pin 12 (paper-out/paper-end = bit 5 of the control register) of the parallel port LPT1 (957 = 0x03BD is the address of the control register of the parallel port LPT1).

Windows doesn't allow direct access to the parallel port. Driver software has to be installed (i.e. PortTalk).

`/keyfromserial port,baudrate`

Translates input from the serial port into key presses for plot input items (key event).

Example: `/keyfromserial com1,19200` reads from the first serial port at 19200 baud.

It is currently inefficiently implemented since it polls the serial port.

# 12 The Import Format

Treatments and questionnaires can be brought into a readable format with File → Export. This format may also be read in again. This format is not described in detail here. Generally, however, the elements of the stage tree are given a form such as the following:

```
element name {
    option1 = value1;
    option2 = value2;
    subelement1
    subelement2
}
```

The sub elements have the same structure. Exporting a treatment or a questionnaire supplies the options.

# Index

## Symbols

## A

## B