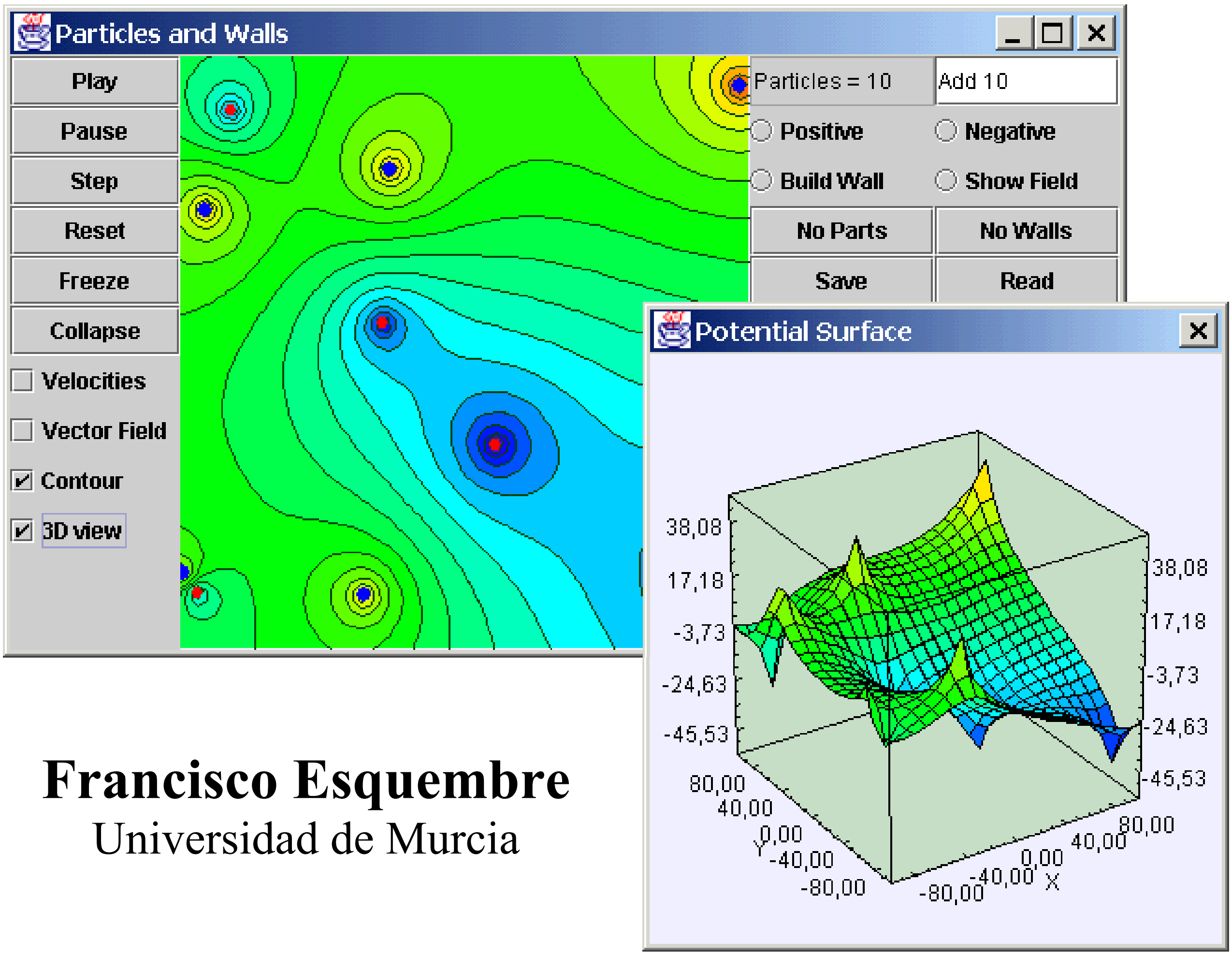
Easy Java Simulations

The manual

for version 3.1



Francisco Esquembre
Universidad de Murcia

Ejs uses *Open Source Physics* tools
by **Wolfgang Christian**

August 2002
http://fem.um.es/Ejs

Foreword

Simulations are playing an increasingly important role in the way we teach or do Science. Specially in Education, where computers are being used more and more as a way to turn lectures more attractive to students and to help achieve a deeper understanding on the subject being taught.

However, it can not be said that computer simulations are used by most of our teachers and trainers. In many cases, this is due to the fact that teachers are reluctant to use a technology they do not fully understand or control. In many others, that they do not find a product that completely meets their educational needs.

A good solution to both problems is to help teachers create their own simulations. We have found that, by creating a simulation, many teachers get a new perspective of the phenomenon they are trying to explain, which almost always increases their enthusiasm about the use of this technology with their students.

An alternative approach, and a very promising one, is to let students create their own simulations, thus engaging in what is called by educational researchers *constructive modeling*. This has the advantage of getting the student to *do* science in an exploratory and constructive way, achieving many of the recommended best-practices in the classroom.

Creating a simulation by oneself requires an extra effort, this is the truth. The starting point, and this is the important part, is a full understanding of the phenomenon being simulated. From this, some technicalities are needed to express the behavior of the phenomenon in computer form.

This manual and the software it describes, **Easy Java Simulations** (**Ejs** for short), address this problem. Both have been specifically designed to teach how to create scientific simulations in Java, in a quick and simple way.

Their audience are science students, teachers or researchers who have a basic knowledge of programming computers, but who cannot afford the big investment of time needed to create a complete graphical simulation.

They may be able to describe the models of phenomena of their respective disciplines in terms of algorithms of a computer language, but still need an extra effort to create a sophisticated, interactive graphical user interface, in the style of simulations and software programs one can find nowadays in the Internet.

With this situation in mind, **Ejs** has been designed to help a person who wants to create a simulation, concentrate most of his/her time in writing and refining the algorithms of the underlying scientific model (which is his/her real expertise) and dedicate the minimum possible amount of time to the programming techniques. And by doing so, still obtain an independent, high performance, Internet-aware, final product.

This manual is structured in two parts. The first one provides a quick starting guide to help you get the basics and have you up and running pretty quickly. The second one goes through a deeper coverage of all the concepts involved in creating a simulation as well as thoroughly describes **Ejs** interface and capabilities.

The final goal is to teach you how to structure a simulation so that it can be correctly translated into a computer program, and instruct you how to use **Ejs** to implement this structure into a Java applet, including instructions to distribute it through the Internet.

The choice of Java as development language is justified in terms of its wide acceptance by the international Internet community, and the fact that it is supported under several software platforms. This implies that **Ejs**, and the simulations created using it, can be used as independent programs under different operating systems, or be distributed via Internet and run within html pages by most popular web browsers.

Contents

PART I. Getting Started

This page intentionally left blank

1

1. Introduction

1.1 What is Easy Java Simulations?

Easy Java Simulations (from now on, **Ejs**) is a software tool designed for the creation of discrete computer simulations.

A discrete computer simulation, or simply a computer simulation, is a computer program that tries to reproduce, for pedagogical or scientific purposes, a natural phenomenon through the visualization of the different states that it can have. Each of these states is described by a set of variables that change in time due to the iteration of a given algorithm.

We shall describe, later in this manual, the structure of a simulation in detail. For the moment, this suffices to learn that **Ejs** is a program that helps you create other programs; more precisely, scientific simulations.

There exist many programs that help create other programs. The most basic ones are called compilers; the most complete are the tools for visual programming, very popular in recent years. **Ejs** would fit into the category of the so-called *code generators*.

But what makes **Ejs** different from most other products is that **Ejs** is not designed to make life easier for professional programmers, but has been conceived by science teachers, for science teachers and students. That is, for people who, like you and me, are more interested in the content of the simulation, the simulated phenomenon itself, and much less in the technical aspects needed to build the simulation.

Hence, **Ejs** provides a conceptual structure and simplified tools that allow concentrate most of your time in the description of the model of the phenomenon you want to simulate.

Nevertheless, the final result, which is automatically generated by **Ejs** from your description, could, in terms of efficiency and sophistication, be taken as the creation of a professional programmer. In particular, **Ejs** creates Java applets that are independent, multi-platform, which can be visualized using any

Web browser (and therefore distributed through the Internet), read data across the net and be controlled using scripts from within html pages.

Because there is an educational value in the process of creating a simulation, **Ejs** can also be used as a pedagogical tool itself. With it, you can ask your students to create a simulation by themselves, perhaps by following some guidelines provided by you. This way, **Ejs** can help your students make their conceptions explicit. Used in groups, it can also improve your students' capabilities to discuss and communicate about science.

1.2 How to use this manual

Do not worry if this manual looks rather thick when it comes out of your printer. You don't need to read it from cover to cover to start creating your simulations.

To start working with **Ejs**, you should definitely complete part I. This will instruct you how to install and run **Ejs** in your computer and will guide you through a step-by-step example of creation of a simple simulation. After reading part I, you will have a general feeling about how to use **Ejs** and, depending on your degree of expertise with computers in general, you could be able to generate a second simulation on your own. Chapter 4. in part I, finally tells you how to distribute your simulation to others or publish it on the Internet (if you have a web server at hand).

If, later on, you want to get a full description of the functionality of **Ejs,** then you should also read part II. Finally, the appendices contain information intended to be used as reference only. That is, you will not read them until you need a particular information, then you'll just go there and search for this information.

If, on the other hand, you are an expert user of computers, or you have programming experience, you may choose to skip part I and get directly to the detailed description of **Ejs**.

It is always advisable to have a copy of **Ejs** running in your computer when you read this manual, so that you can try the examples on your own and see that they actually work.

Have a good time and enjoy **Easy Java Simulations**!

1.3 Acknowledments

Easy Java Simulations, now in version 3.1, is the result of a project that has been carried out for several years and under different conceptions and implementations. For this reason it owes a lot to contributions from several groups of people.

The first group I would like to mention is that of my colleagues at the University of Murcia, Spain, Professors Ernesto Martín and Jose Miguel Zamarro who, as members of the Colos group in Murcia (http://colos3.fcu.um.es), always shared their interest and enthusiasm with me for this project.

The second group of people is the whole Colos community (http://www.colos.org), a group that originated in 1989 in a project founded by the (then called) European Community, that evolved autonomously from then, and that now comprises members from about seventeen universities and institutions from all over the world.

I am also very grateful to Vincent Boland, Melissa Cheung, Jee Park and James Sulzen, the group of master students from Stanford University, California, who, in the summer of 2000, took a previous version of **Ejs** as their case study for the interactive media project they had to conduct (http://ldt.stanford.edu/~mcheung/MelissaCheung_portfolio/EJS.html) as part of their Master studies under Professor Decker Walker of the School of Education of this university.

Following this chronological order, but with special emphasis, I am in great debt to Professor Wolfgang Christian of Davidson College, North Carolina, who introduced me to the development of both his Physlets (http://webphysics.davidson.edu/Applets/Applets.html) and his more recent work in the Open Source Physics project (http://www.opensourcephysics.org). In fact, release 3.0 of **Ejs**, whose main innovation was on the simplification of the construction of the view of a simulation, was possible only thanks to the use of the new display Java classes that originated in this project.

And last, but not least, I am also very grateful to Professor Fu-Kwun Hwang, of the National Taiwan Normal University, Taiwan, (http://) for his interest and support for **Ejs** and also for the joint work in the development of new educational web services based on **Ejs**.

A special place of honor is reserved for my family: my wife Araceli and my daughters Araceli and Maria Belén, for all those weekend evenings that I spent, and still spend!, in front of the computer.

This page intentionally left blank

2. Before we start

Before we start working with **Ejs**, let us make sure that you have everything we need.

2.1 Documentation

This manual is the right place to begin with. It is (so far) the one and only written description of this version of **Ejs**. If you need an updated copy of it, go to the official web pages for **Ejs** in http://fem.um.es/Ejs. You will also find there links to examples, software updates and any other source of information that might appear.

This manual will tell you how to obtain and install the software you need, will guide you through the basics of **Ejs** and also get you to the secrets of it.

There is a second source for information, which are the examples bundled with the distribution of **Ejs**. It is very convenient that, after getting familiarized with **Ejs**, you take a look at these examples. The directory *_examples* in your working directory[1] contains complete, non-trivial simulations that can help you find how to do certain things that you may be interested in doing, and perhaps serve as source of inspiration for your own simulations.

Finally, you should also have a look at **Ejs** official web pages (see the address above) under the link *Examples*. They contain some other interesting examples created either by the author or by other users of **Ejs**. When you finish your simulation, don't forget to send it to the author (that is, me) at the e-mail address fem@um.es for inclusion in this gallery of examples!

2.2 Installation Instructions

You also need the software, of course. If someone installed **Ejs** for you in your computer, then you (have a good friend and) can proceed to the next section. If this is not the case, here go the necessary installation instructions.

[1] Learn in section 2.3 how to locate it in your disk.

Before even thinking in installing **Ejs** you must have the **Java Software Development Kit** (Java SDK) installed in your system. Latest version I tried to work with, and never caused me any kind of problems at the moment of this writing, is version 1.3, which I therefore recommend[2]. For this, please obtain Java SDK free distribution file from ***Sun*** at its web site http://java.sun.com, and run the standard installation procedure for it.

This procedure is very simple, just double-click the (rather big) installation file and answer *Yes* to every question it asks.

During the installation[3] of Java SDK, you are offered to select an installation directory, which defaults to *c:\jdk1.3* (maybe *c:\jdk1.3_xxx* if it is the latest release of this version). Unless you have special requirements, I recommend that you accept this.

Now, installing **Ejs** simply consists in uncompressing[4] the distribution file *Ejs.zip* in any directory you want; *c:\Ejs* or *c:\Ejs3.1* are both good choices.

> Because some Java plug-ins (for instance, jdk 1.4) can work incorrectly with file or directory names which contain white spaces, I recommend choosing a directory that has no white spaces in its name, nor in its path, as installation directory for **Ejs**. For instance, *C:\My Ejs* or *C:\My documents\Ejs* are not good choices.

The file *Ejs.zip* can be obtained, once more, from the official web pages indicated above.

If you got **Ejs** from a CD then perhaps you don't need to uncompress the file and you have already a directory in the CD named *Ejs*. If this is the case, copy this directory to any place in your hard disk. The root directory *c:\* is a good place, but you may choose any other that suits your hard disk organization.

IMPORTANT NOTICE: If, and only if, you chose an installation directory for Java SDK different from the default *c:\jdk1.3*, then you must edit the *batch*[5]

[2] Yes, I have tried version 1.4 of the SDK. Although it works well most of the time, I have found some strange behaviour in its plug-in for web browsers, which quits the browser (after a second or so) when displaying applets that use the Mid-point or Runge-Kutta algorithm for solving differential equations… You can still decide to use sdk 1.4, but you have been warned ☺.

[3] The description of the installation procedure is given for the Windows family of operating systems. However, **Ejs** (and Java SDK) works under Windows 98/ME/2000/XP, Linux and MacOS.

[4] If you don't have an appropriate uncompressing utility, get one from http://www.winzip.com/. They have free evaluation versions that can do the job for you, too. The compresing scheme may vary under a different operating system.

[5] A batch file is just a text file which contains a sequence of operating system commands. You can edit it with your favorite editing tool (in *Windows* you could use *Notepad*, for instance).

file *Ejs.bat*, which you will find in the directory where you installed **Ejs** and change the first line, which reads now

```
set JAVAROOT=c:\jdk1.3
```

and substitute *c:\jdk1.3* for whatever directory you used when installing Java SDK.

If you want to run **Ejs** in a language other than English (so far, a Spanish version is available and a Chinese one is in preparation), you have to use a slightly different batch file which will be named *Ejs_xx.bat*, with *xx* standing for an abbreviation of your language (for instance, the Spanish version is called *Ejs_es.bat,* the Chinese – actually, Taiwanese - version is *Ejs_zh.bat*. If this is the case, whenever you see *Ejs.bat* written in this manual, you have to read your *Ejs_xx.bat* file.

And this is all you need, **Ejs** should be ready to run. However it is advisable that you read next section before running **Ejs**.

2.3 Organizational information

If you inspect the directory where you installed **Ejs**, you will find a batch file *Ejs.bat* (and perhaps other files of the type *Ejs_xx.bat*) and two directories: *data* and *Simulations*. The first directory contains specific data required to run **Ejs**. Unless you are an expert with Java, don't touch it (and even if you are an expert, there is no reason to touch it!).

The directory *Simulations* is the one where you will work when running **Ejs**. It contains three directories: *_data, _examples* and *_library*[6].

The fact that the three directories in your working directory start with an underscore character '_', is **Ejs**' particular way of making sure that it will not interfere with your future work. When, later on, you create your own simulations, you should make sure never to use names that start with an underscore character[7].

The first two directories contain some samples simulation files (in *_examples*) included in the distribution and the data they need to run (in *_data*). They are not really neccesary to run **Ejs**. They are there just for your perusal, so you can read, inspect, modify and run these files or even delete the whole directories.

The case of the *_library* directory is different. It is needed both to run **Ejs** and the simulations that you will generate with it. So, please don't ever move, rename or delete it.

[6] Release note: This is a small change from version 3.0. I decided to put the jar library *_ejsLibrary.jar* in a directory in order to help clean the simulations directory from time to time. There is a second technical reason for this change which relates to the possibility of adding new libraries to the distribution in the future.

[7] There is however an exception to this rule, which is the case when you want to create a file to be merged with other. We will describe this case in detail in part II.

The typical operation is to run **Ejs** in your *Simulations* directory (the default starting point) and generate your files and simulations in it. However, you can choose to work in any other directory that you want. This can be useful if you share the computer or the disk space with another user, for instance.

If this is the case, then you have to copy[8] the *_library* directory to your new working directory and edit the batch file *Ejs.bat*, which we referred to in the previous section. The second line of this batch file reads now

```
set EjsDir=Simulations
```

You need to change it so that *EjsDir* points to your new working directory, wherever it is.

This is most likely the last modification that you may want to do on your *Ejs.bat* file. Of course, you may want to have different *Ejs.bat* files, which use different working directories, either for different users or for different tasks.

> Soon, as you create simulations in the *Simulations* directory, either by your own or by trying the examples, this directory will become populated, even crowded. You can safely clear this directory by removing any file that you no longer need (but be sure that you **no longer** need it!). Just respect the original directories: recall that they start with an underscore character '_'.

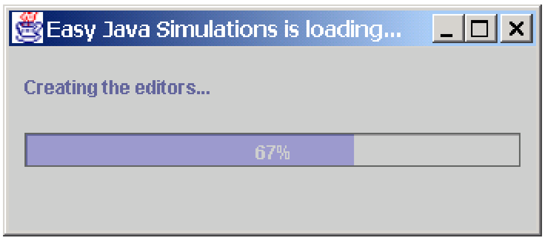
2.4 Running Easy Java Simulations

2.4.1 Launching the program

It is time to start **Ejs**! I assume that you installed it successfully in, say, the directory *c:\Ejs*. We want to execute the file *Ejs.bat*[9] which is placed in this directory. Under *Windows*, for instance, you do this either using *Windows Explorer* and double-clicking on this file or with the *Start→Run* option of the *Windows* menu and typing *C:\Ejs\Ejs.bat* in its *Open* text field.

> Under Linux of MacOS X you need to open a shell window and execute the shell scripts *Ejs.linux* and *Ejs.macosx,* respectively, included with the distribution.

After some seconds, during which you will see a progress dialog like this one,



Easy Java Simulations is loading...

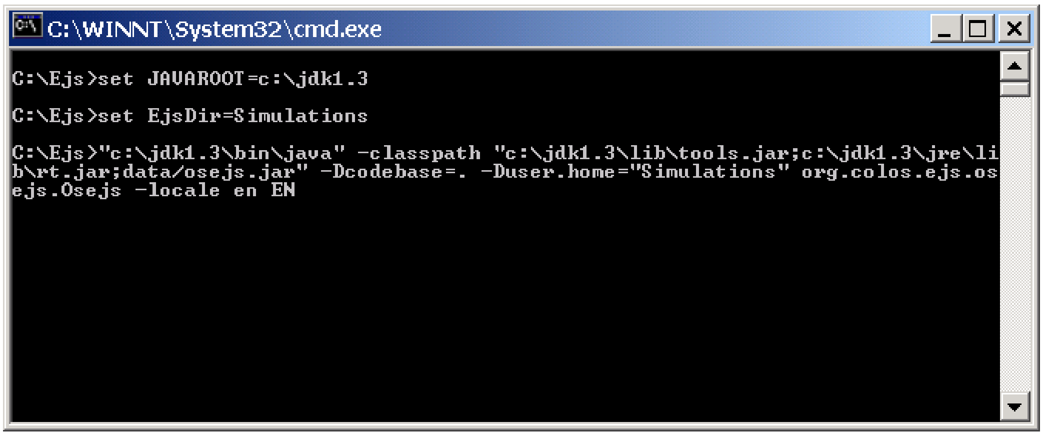
Creating the editors...

67%

[8] We suggest *copying* better than *moving*. It is a good idea to leave a copy of this library diretory in its original place, so that other users can find it. But you could, in principle, delete the original one after the copy.

[9] Recall to read' *Ejs_xx.bat* if you are using a different language.

you will get two windows, a black text window with some written lines in it, similar to this,

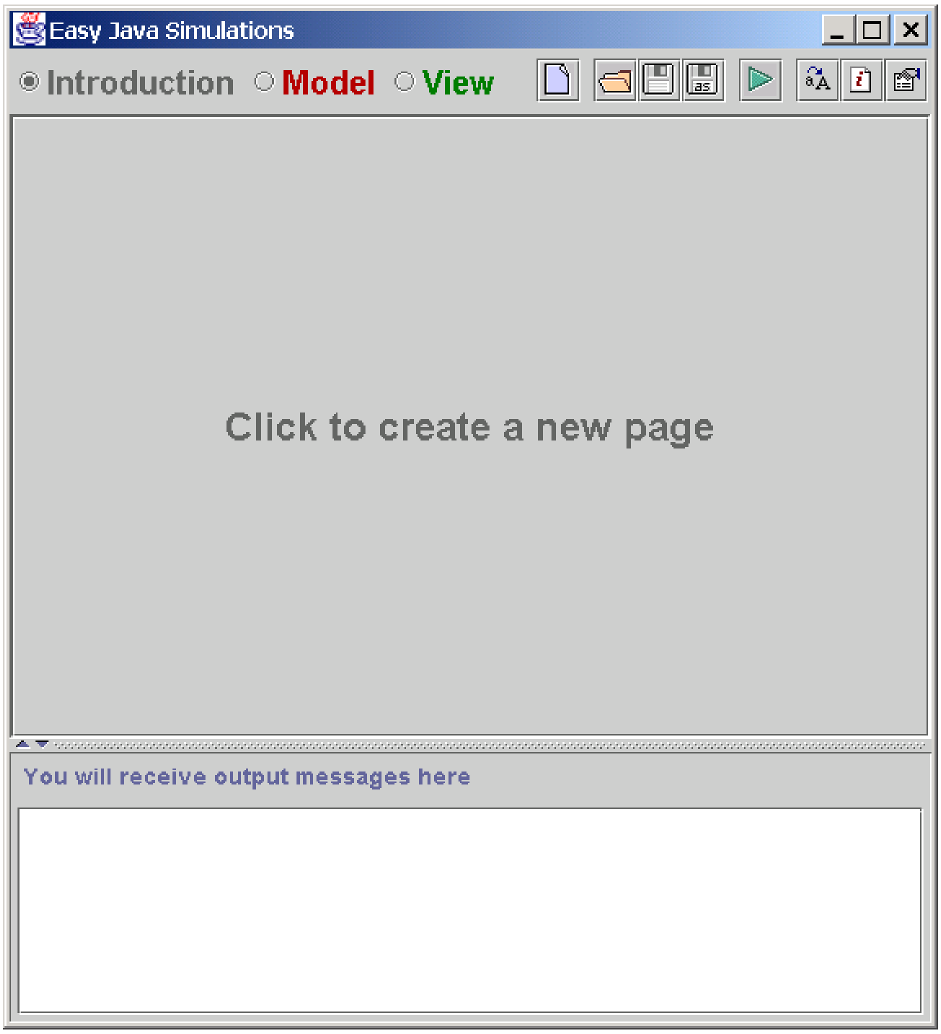


```
C:\Ejs>set JAVAROOT=c:\jdk1.3

C:\Ejs>set EjsDir=Simulations

C:\Ejs>"c:\jdk1.3\bin\java" -classpath "c:\jdk1.3\lib\tools.jar;c:\jdk1.3\jre\li
b\rt.jar;data/osejs.jar" -Dcodebase=. -Duser.home="Simulations" org.colos.ejs.os
ejs.Osejs -locale en EN
```

and a second one which is the real **Ejs** interface[10]:


Easy Java Simulations
● Introduction
○ Model
○ View
Click to create a new page
You will receive output messages here

[10] Release note: The interface is the same as for version 3.0, except that a new icon has been added to the right-most end of the toolbat. It is the one that brings in the 'Options' dialog: ; its functionality will be described later.

The pointing hand is obviously not part of the interface, I have added it myself, in some places along this manual, to help me focus your attention to particular points.

We will ignore the text window (if you want, you can minimize it, but do NOT close it!) and will pay attention only to **Ejs** window.

On this window, at the top (just where the hand is pointing to), we see three radio buttons to the left and a toolbar with some icons to the rightmost end of the window. We will use some of these icons soon.

The three radio buttons are labeled *Introduction*, *Model* and *View*, corresponding to the three parts of any simulation created with **Ejs**, which we will describe later.

These radio buttons work in such a way that only one of them can be selected at a given moment. Right now, the *Introduction* button is selected (which makes its circle appear filled).

Below this line, we see an empty area with an inviting message written using the same color as the *Introduction* label.

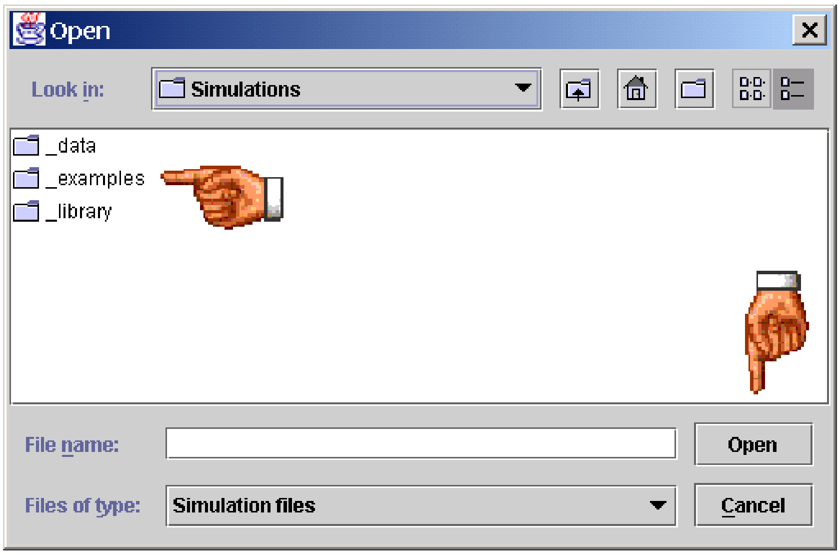
Finally, at the bottom, we see a message area that **Ejs** will use to display messages to you as result of your actions. This message area has an independent, different color.

This color system means that the whole middle panel (everything which has the labels in the same color) corresponds to the *Introduction* top button, and you can bet that if you click on any of the other two radio buttons, *Model* or *View*, the color and the aspect of the middle part of **Ejs** interface will change.

And this is enough for now. We will close the section and the chapter just testing the installation: reading a sample simulation file and running it.

2.4.2 Reading a sample simulation

Please, proceed on your computer as follows. First, click on the 📂 icon in the toolbar. A new window (an open file dialog) will appear.



Open

Look in: Simulations

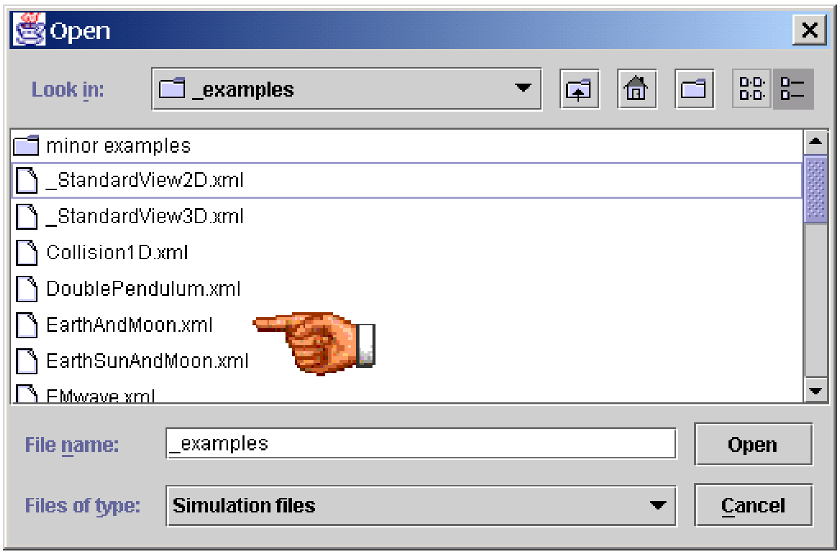
_data
_examples
_library

File name:

Files of type: Simulation files

Yes, this is **Ejs** working directory. Double-click (that is, click twice quickly) on the *_examples* directory. Alternatively, you can click once on *_examples* and then click on the button labeled *Open*. Both are pointed to by the hand, in the picture above.

The dialog will show what is inside it.



Open

Look in: _examples

minor examples
_StandardView2D.xml
_StandardView3D.xml
Collision1D.xml
DoublePendulum.xml
EarthAndMoon.xml
EarthSunAndMoon.xml
EMwave.xml

File name: _examples

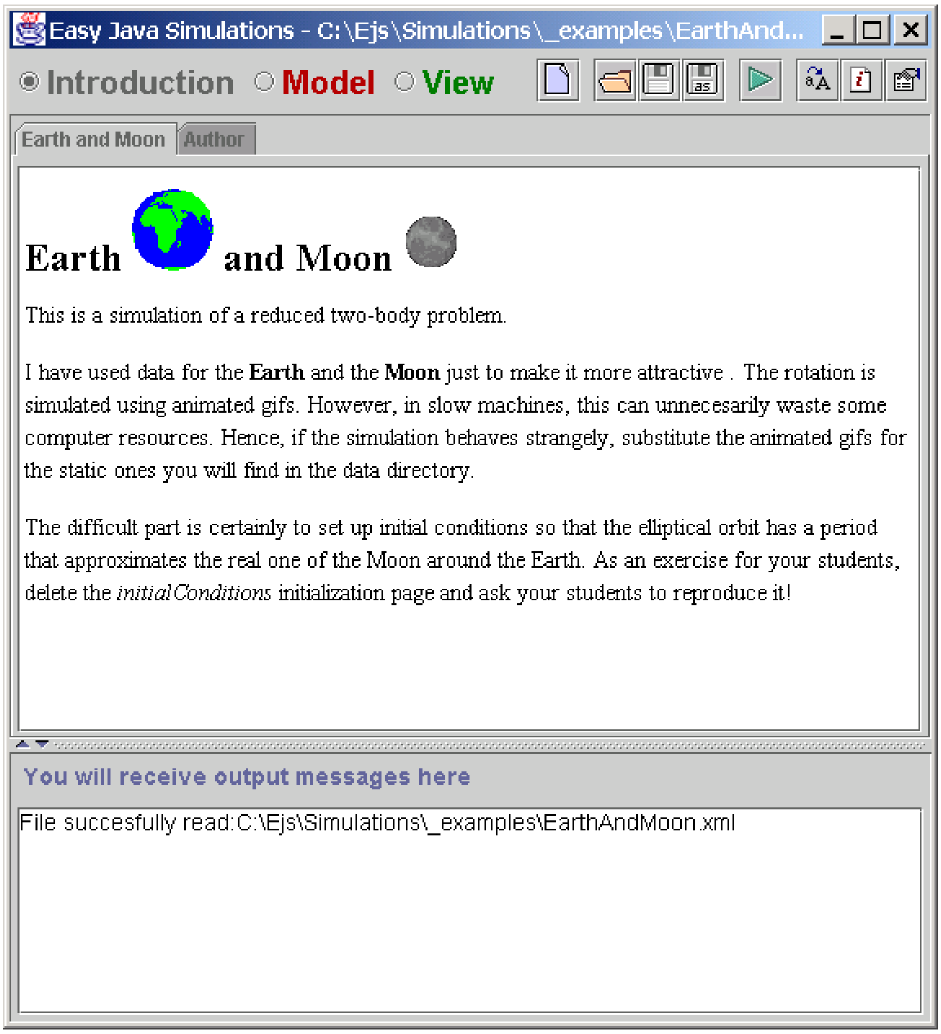
Files of type: Simulation files

This is the list of files with the extension *.xml* (the extension for files generated by **Ejs**) in this directory.

> The number and the names of the sample files that you will actually find in your directory may be different from what is shown in this manual. The reason is that I may have added new examples from the time I took this picture.

Select, for instance, the file named *EarthAndMoon.xml* by double-clicking on its name (again, you can click once on its name and then click on the *Open* button).

Ejs will load the file.


Easy Java Simulations - C:\Ejs\Simulations_examples\EarthAnd...
● Introduction ○ Model ○ View
Earth and Moon
Author

Earth **and Moon**

This is a simulation of a reduced two-body problem.

I have used data for the **Earth** and the **Moon** just to make it more attractive . The rotation is simulated using animated gifs. However, in slow machines, this can unnecesarily waste some computer resources. Hence, if the simulation behaves strangely, substitute the animated gifs for the static ones you will find in the data directory.

The difficult part is certainly to set up initial conditions so that the elliptical orbit has a period that approximates the real one of the Moon around the Earth. As an exercise for your students, delete the *initialConditions* initialization page and ask your students to reproduce it!

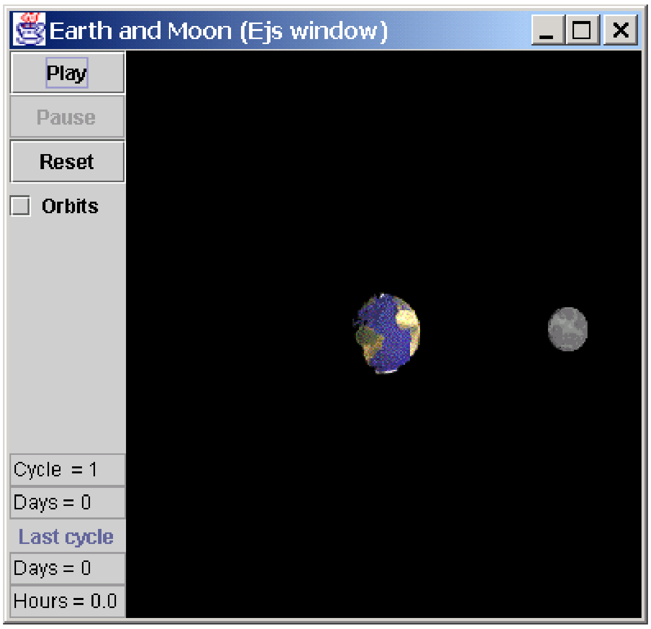
You will receive output messages here

```
File succesfully read:C:\Ejs\Simulations\_examples\EarthAndMoon.xml
```

Notice that the *Introduction* panel (which occupies the middle area) is not empty anymore and that the message window displays a message confirming the success of the reading process.

You can click on the different radio buttons and tabs to browse how the file describes the simulation. You may not understand everything that what you will see, but **Ejs** seems to be working properly.

Depending on how I saved the file before distributing it, a separate window may also appear, showing the view of the simulation: a black area in space with two images, the Earth and the Moon, and a control panel to its right.



Notice that what we have read is the definition of the simulation, but the simulation itself is not running. Hence, the images do not move. Notice too that the title of the window also tells us that it is a window created by Ejs. The final simulation will not display this notice.

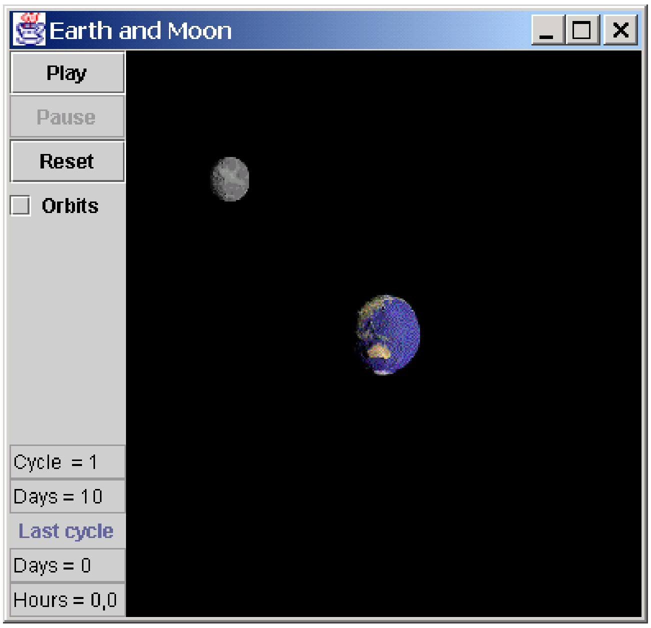
To finally check correct operation, let us actually run the simulation.

2.4.3 Running the example

This is done by clicking on the icon ▷. Please, do it now. After a few seconds, **Ejs** outputs the following reassuring message

```
Generating simulation file C:\Ejs\Simulations\EarthAndMoon.java ... Ok!
Creating jar file C:\Ejs\Simulations\earthAndMoon.jar ... Ok!
Congratulations! The simulation was generated successfully.
Trying to run simulation C:\Ejs\Simulations\EarthAndMoon.bat...
```

And a new window (with the Earth and the Moon now in motion) appears on the screen.



If this is what happened to you, it means that the simulation was successfully created. If you look into the directory *Simulations,* you will see that there are a few new files, all of them with a name that starts with either *EarthAndMoon* or *earthAndMoon*. A description of what these files are will be given in chapter 4.

> If the simulation contained some errors (I would have never included it in my _*examples* directories, to begin with! ☺) then some, perhaps a lot of, complaint messages from the compiler would appear in the message area of **Ejs**. We will also learn how to deal with them in this manual.

If you want, you can play with this simulation, for instance clicking on the *Reset* or *Orbits* buttons. But for our purposes, it is enough. If everything went on for you as described here, you can be sure that your system is properly installed.

If you now close the simulation (just close its window as you close any other window in your operating system[11]), **Ejs** finally says

```
Congratulations! The simulation seems to run alright.
You can also run the simulation from the generated html page
```

And this is the end of the chapter. In it, you have learnt how to install **Ejs**, and how to read and run any of the examples that come with the distribution of **Ejs**. It is time now for you to create your first simulation. Please, move on to the next chapter!

[11] In Windows, by clicking on the right-top button, the one that displays an X.

3. A first complete example

Before we get into details of how to correctly structure a simulation and describe all the possibilities of **Ejs**, we will first create a complete working simulation. This way, you will later be able to identify the parts of a simulation with respect to what you did in this example.

This is also a small trick on my side. By letting you create a simulation in the first chapters, you will realize that it is really simpler than you expected (with the help of **Ejs**, of course ☺), and you will always tell your colleagues and friends that this manual is so wonderful that you were able to create a simulation at the very start!

The example we will build relates to Lissajous' figures. Do not worry if you don't know very much about them or if you don't completely understand everything you will be doing. The main objective is that you go through the full process and get a general perspective of it.

3.1 The structure of a simulation

We start with a little bit of theory. Most computer simulations of scientific phenomena can be described in terms of the **model-control-view** paradigm.

This paradigm states that a simulation is composed of three parts:

- The **model**, which describes the phenomenon under study in terms of
  - variables, that hold the different possible states of the phenomenon,
  - interrelationships among these variables (corresponding to the laws that govern the phenomenon), expressed by computer algorithms.

- The **control**, which defines certain actions that a user can perform on the simulation.

- The **view**, which shows a graphical representation of the different states that the phenomenon can have. This representation can be done in a realistic or schematic form, but in such a way that the user appreciates the most relevant aspects of the simulated phenomenon.

These three parts are deeply interconnected. The model obviously affects the view, since a change in the state of the model must be made graphically evident to the user. The control affects the model because control actions can (and usually do) modify the value of variables of the model. Finally, the view affects the model and the control, because the graphical interface can contain components that allow the user to modify variables or perform the predefined actions.

In fact, going a step further in the process of simplifying the construction of a simulation, **Ejs** suppresses the control part, merging it half into the view, half into the model. Actually, modern computer programs are interactive, which means that the user can modify the program's logic by doing some gestures (such as clicking or dragging the mouse, or hitting the keyboard) with the computer peripherals on the program's interface (or view). Thus, the view itself can be used to control the simulation.

On the other hand, if we want this interaction to have certain relevance on the program, these gestures on the interface need to trigger actions that affect model's variables. Therefore, the best place to define these actions is in the model itself.

Creating a simulation in Ejs consists in defining its model and its view, establishing the mutual connections needed for

a) the correct visualization of the state of the phenomenon being simulated and

b) the appropriate interaction of the user with the view (either to modify this state or to perform the actions defined on the model).

This explicit separation in parts reinforces conceptually the central role of the model of a simulation. It is the model which defines what the program simulates and how. Notice also that there may be different views for a given model.

It also makes the task of creating a simulation more modular and promotes reusability, since the parts can be created independently in time, or by different people.

Finally, and though it is not really part of a simulation, it is always convenient to include some textual description of what a simulation does, or instructions on how to operate with it, specially if we plan to use it with others, students or colleagues, for instance. This is the purpose of the extra part, which I have placed before the others, that is labeled *Introduction*.

This way, we finally obtain the three radio buttons in **Ejs** interface: *Introduction*, *Model* and *View*.

3.2 Lissajous' figures

We are now ready for our example. Let us state what we will simulate.

When we superpose two simple harmonic movements with perpendicular directions, we obtain a planar movement that is described by the equations

$$x = A_1 \cos \omega_1 t$$
$$y = A_2 \cos(\omega_2 t + \delta)$$

where the A_i's denote the amplitudes of the respective movements (horizontal the first one, vertical the second), the ω_ι's denote the respective frequencies and δ denotes the phase delay between both movements.

If we supply these two signals to the horizontal and vertical inputs of an oscilloscope, its beam will describe a movement that is the result of the superposition of both individual movements and that can adopt several nice figures, depending on the value of the ratio $\frac{\omega_2}{\omega_1}$ and of δ.

These curves are called **Lissajous' figures** and are specially nice for the values $\frac{\omega_2}{\omega_1} = \frac{1}{1}, \frac{1}{2}, \frac{1}{3}, \frac{2}{3}, \frac{3}{4}, \frac{3}{5}, \frac{4}{5}, \frac{5}{6}$ and $\delta = 0, \frac{\pi}{4}, \frac{\pi}{2}, \frac{3\pi}{4}, \pi$.

When the beam hits the screen of the oscilloscope, a dot is displayed. For a single beam to describe a full figure we need a screen that has some kind of memory. That is, that gives the dot the occasion to leave a trace.

Hence, **our task consists in building a simulation where two variables *x* and *y* follow the model described above, and which displays the point *(x,y)* in a screen which has 'memory'**. We will also want to be able to modify the values of the frequencies and of the phase delay (the amplitudes only affect the relative size, but not the aspect of the curves), thus visualizing several different Lissajous' figures.

As said in the previous section, to actually build the simulation we need to describe its **model** and build its **view**. Writing the mathematical formulae that relate the variables x and y to the parameters (amplitudes, frequencies, phase delay, time) is the model of our simulation. Displaying it in a computer window is obviously the view. We will also define in the model some actions that will set some combinations of the parameters that produce especially nice figures.

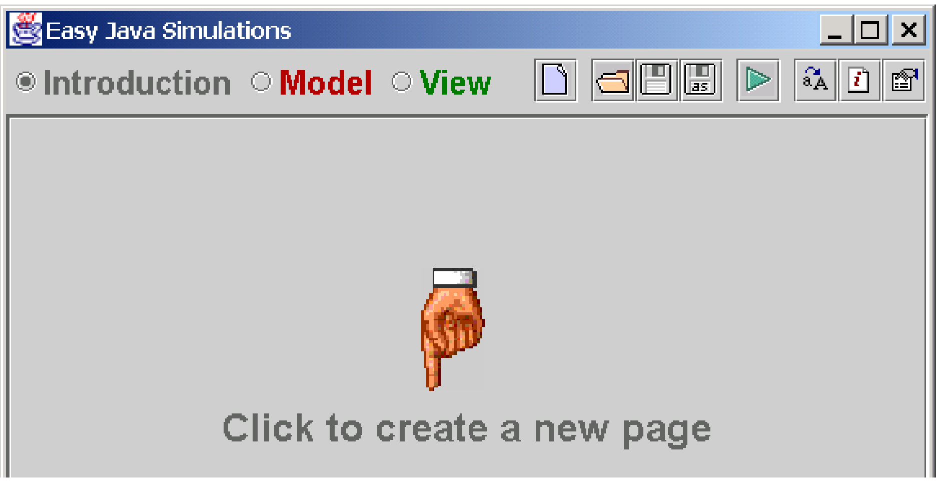
3.3 Writing an introduction

Let's start our work! I assume that you installed **Ejs** successfully and that you know how to start it[12]. Do it now.

If, on the other hand, **Ejs** is already running (as a result of a previous session), you want to set it to a default state, with no simulation on it. In this case, you have to click on the '*Create a new simulation*' icon 🗋.

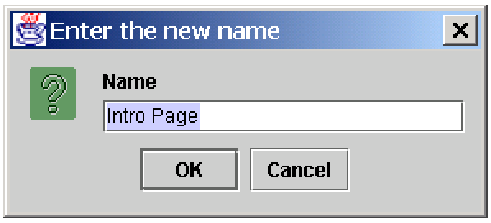
If it is not already selected (its button circle should be filled), click on the *Introduction* radio button (the gray one).

Once in the introduction panel, click on the middle panel that reads '*Click to create a new page*', because this is exactly what we want to do.



As soon as you click on the panel the following dialog will appear.

Because we can create more than one introductory pages, we can give each of them a different name.

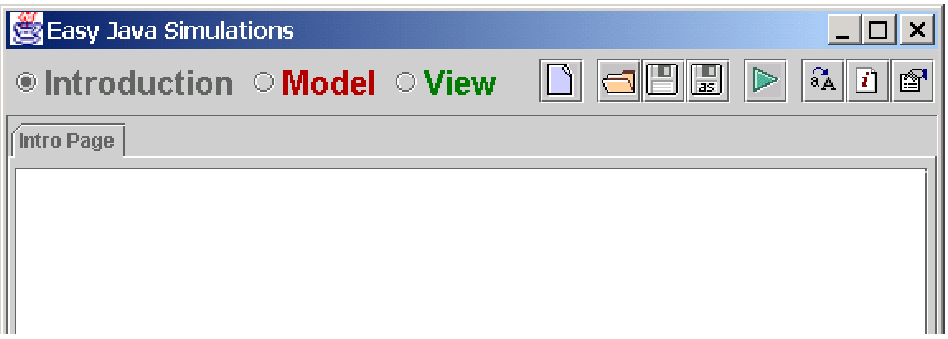


Name: Intro Page

[12] If not, read section 2.4

For the moment, we will simply accept the default name proposed by **Ejs** for this page. So, click on the *Ok* button.

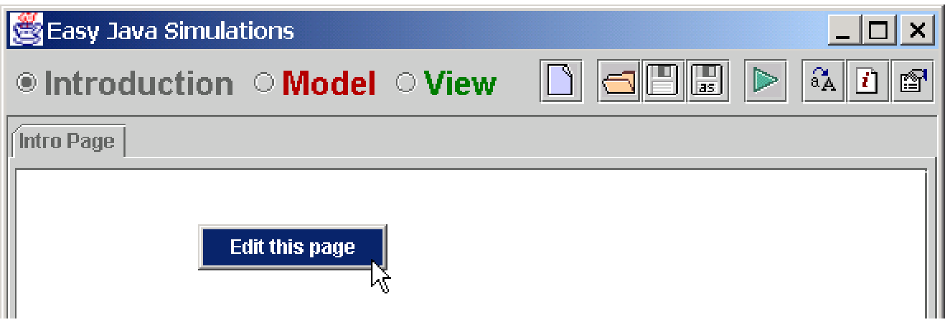
The result looks like this



We will now write our initial introduction in this page. However, we can not write directly into this blank page because this is a special window that is able to display html code, but it only works in read-only mode.

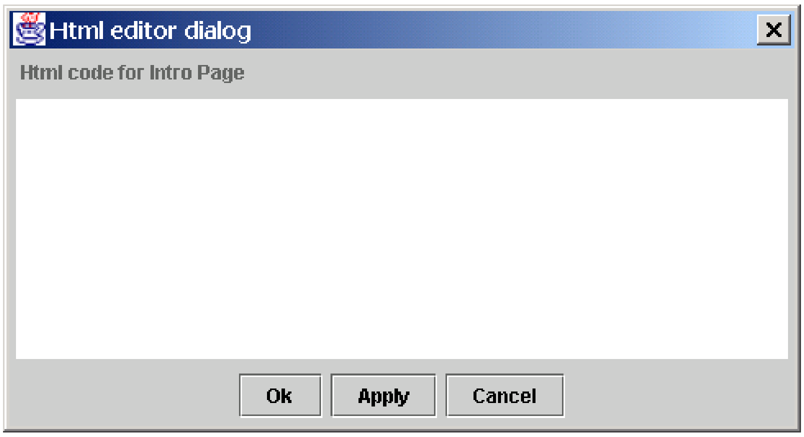
Html code is a special type of text that includes certain simple tags, or extensions, to allow you to control a bit the appearance of what you write [13]. To let you appreciate the importance and utility of html code I will just mention that it is at the heart of most of the pages that you read while navigating on the Internet.

To write our description, we bring-in an editor window. Right-click the mouse (that is, click the right button of the mouse) on the blank window. A tiny pop-up window will appear:



If you select its only option, *Edit this page,* a new window, this time containing an editable text area will appear.

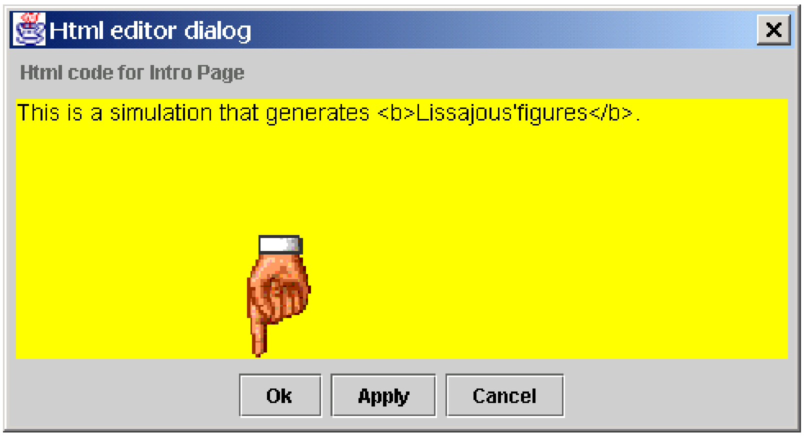
[13] If you are not familiar with html code, you can learn the basics in the appendices.



Click on this empty area and type the following,

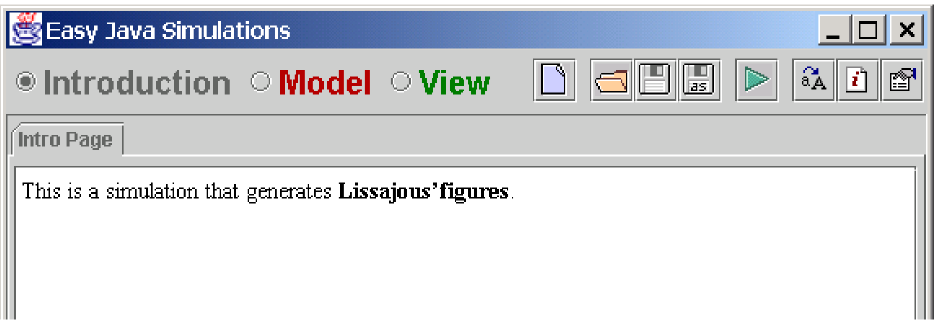
This is a simulation that generates <b>Lissajous'figures</b>.

Notice that, as soon as you start typing, the text area turns its background color from white to yellow. This is **Ejs**' way of telling you that you have modified something, but that this change doesn't apply yet. In fact, if you noticed it, the introduction page is still empty. If you know click on the *Ok* button



This is a simulation that generates <b>Lissajous'figures</b>.

you will see that this window disappears and the introduction page now reflects the change.



This is a simulation that generates **Lissajous'figures**.

Congratulations! If you had never done it before, you wrote your first piece of html code. Probably you noticed the effect that the tags <*b*> and </*b*> had on the display of the text they enclosed: it was written in boldface characters. This is exactly the standard behavior of most html tags. If you want to go a bit further and write a more detailed description, go ahead: edit the page again and add some more html code.

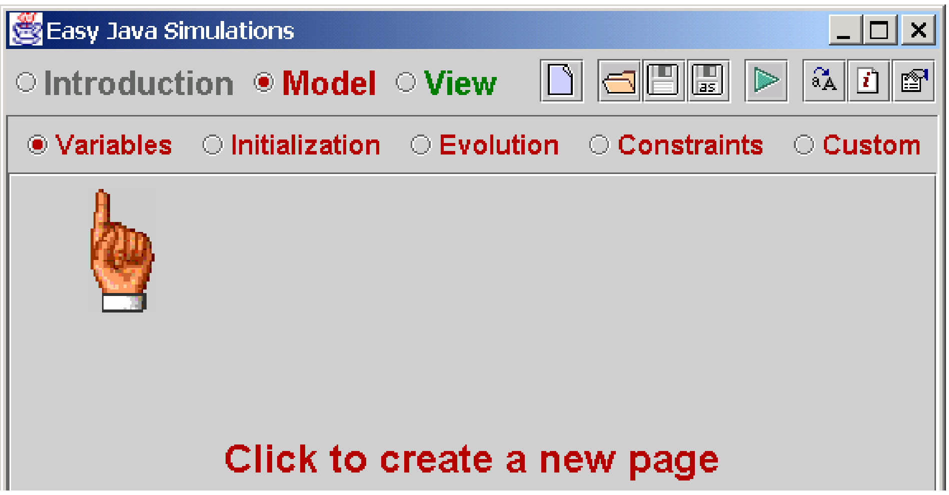
> Although we leave the description of other tags to the corresponding appendix, you can learn a bit from the examples that come with Ejs. A useful tag, that you must definitely know, is the paragraph tag <p>. It does not need a matching </p> enclosing companion (but you can write it, if you want) and indicates the editor to start a new paragraph.

3.4 Building the model

3.4.1 Defining the variables of the model

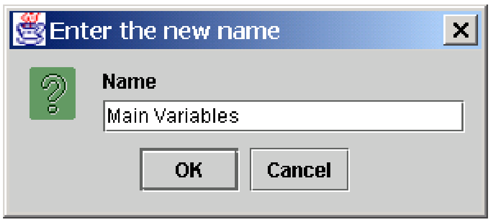
We now proceed to the central part of the simulation, its model. And we start by defining the variables for it.

Click on the *Model* radio button (the red one). Once in the model panel, you will see that it has several subpanels, each with its corresponding radio button.


Easy Java Simulations
○ Introduction ● Model ○ View
● Variables ○ Initialization ○ Evolution ○ Constraints ○ Custom
Click to create a new page

If it is not selected (in the figure it is), select the *Variables* panel.

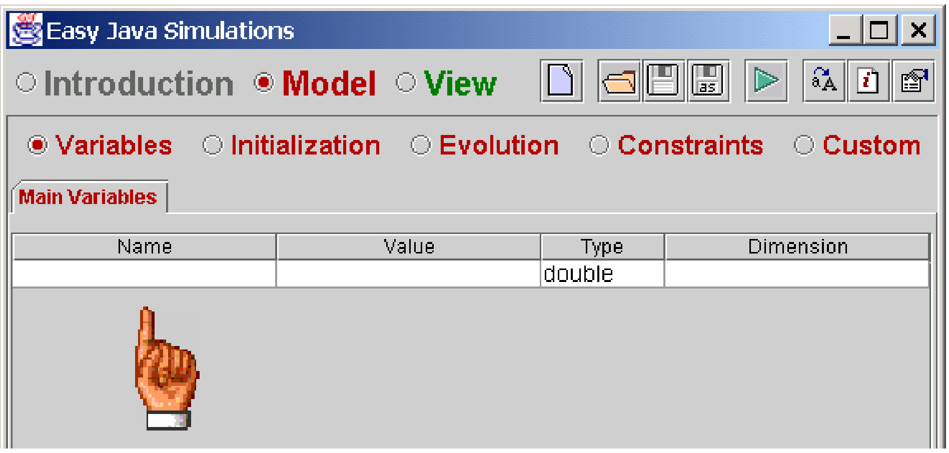
Now, click on the middle empty area to create a new page.



Again a dialog appears to let you enter a name for the new page of variables.

Type, for instance, *Main Variables* and click *Ok* (or hit return).

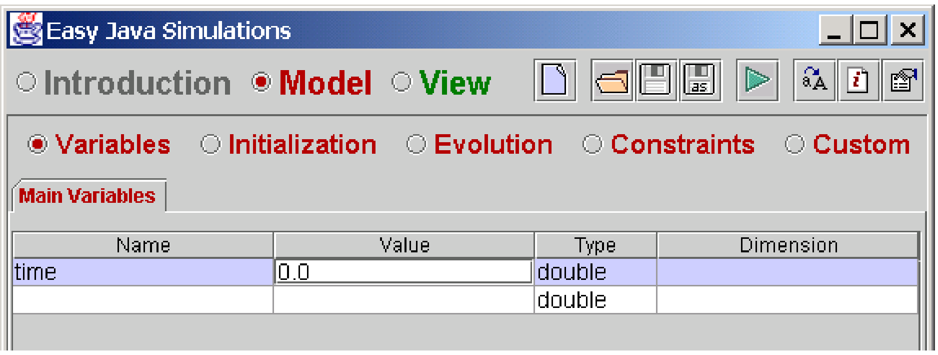
This is how it should look like.


Easy Java Simulations
○ Introduction ● Model ○ View
● Variables ○ Initialization ○ Evolution ○ Constraints ○ Custom
Main Variables

Name	Value	Type	Dimension
		double	

We will now write our variables. Click on the white text field of the column labeled *Name* (the one the hand is pointing to), and type *time*. Then click on the empty field on the column *Value* and type *0.0*. This tells **Ejs** that we want to create a variable called *time* of type *double* with initial value *0.0*, which is acceptable for our purposes. Since this is a simple variable, I mean that it is not an array (this is the Java word for a vector or a matrix), we will leave the column *Dimension* empty.

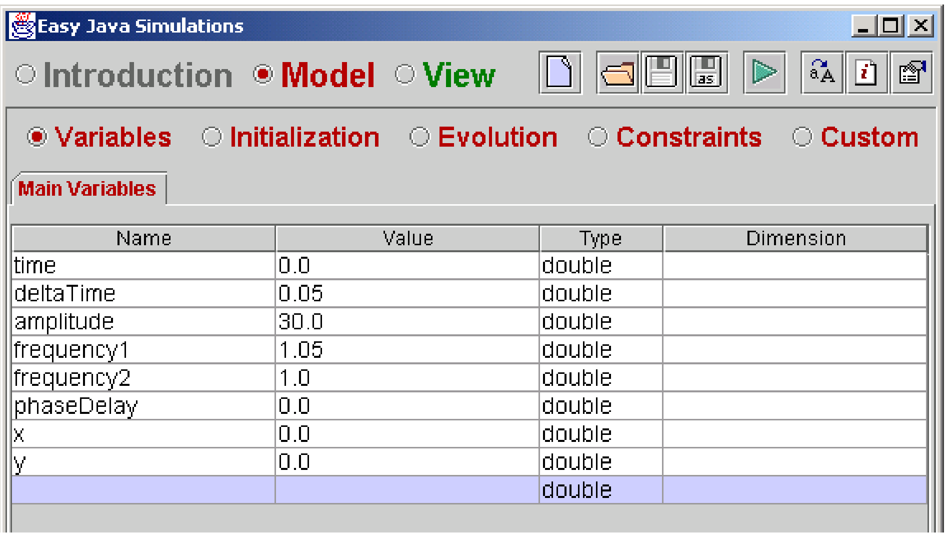
You would have noticed that, as soon as you started typing, the table opened a second row for subsequent variables. So, our table looks now like this.


Easy Java Simulations
○ Introduction ● Model ○ View
● Variables ○ Initialization ○ Evolution ○ Constraints ○ Custom
Main Variables

Name	Value	Type	Dimension
time	0.0	double	
		double	

I know assume that you got the idea of the mechanism and will let you define the double variables *deltaTime*, *amplitude*, *frequency1*, *frequency2*, *phaseDelay*, *x*, and *y*, with initial values *0.05*, *30.0*, *1.05*, *1.0*, *0.0, 0.0* and *0.0*, respectively.

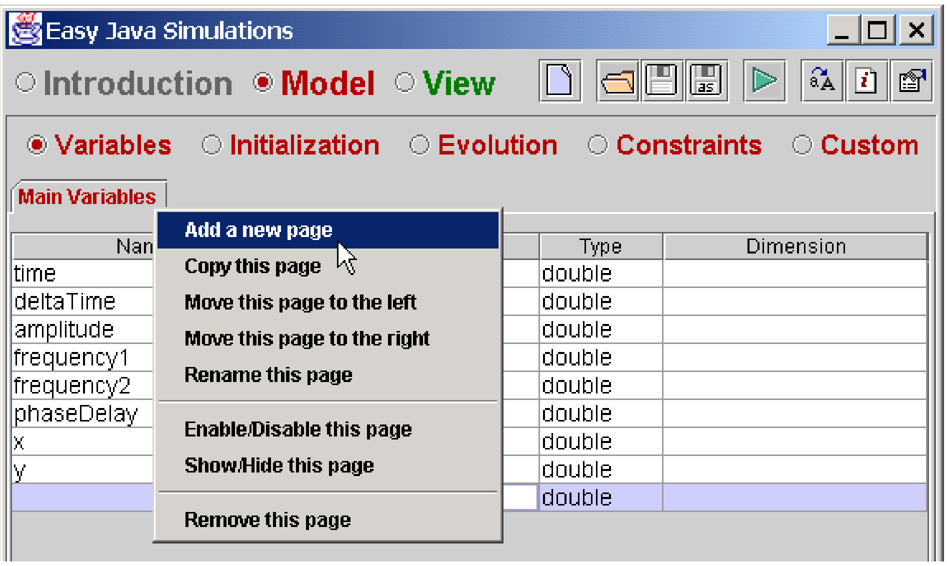
Our final table of variables should look like this


Easy Java Simulations
○ Introduction ● Model ○ View
● Variables ○ Initialization ○ Evolution ○ Constraints ○ Custom
Main Variables

Name	Value	Type	Dimension
time	0.0	double	
deltaTime	0.05	double	
amplitude	30.0	double	
frequency1	1.05	double	
frequency2	1.0	double	
phaseDelay	0.0	double	
x	0.0	double	
y	0.0	double	
		double	

Though these variables completely determine the state for our model, as we will see, we will need some auxiliary variables when we try to configure our view to display it. To create them, but still keep a clear separation with our state variables, we are going to create a second table of variables (i.e. a new page) and will define the new variables there.

Right-click on the top tab of this page (where the label *Main Variables* is displayed) and a popup menu will appear,

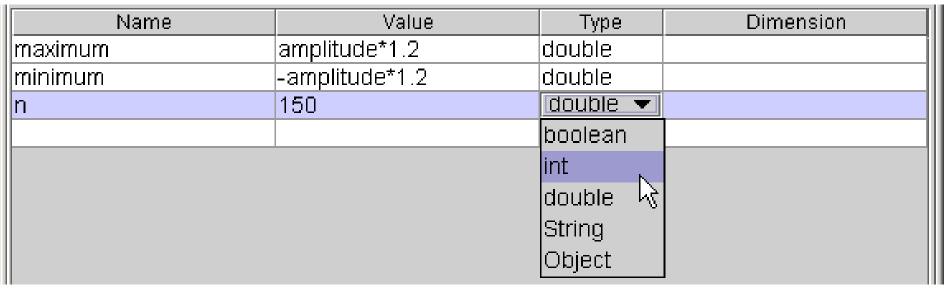

Easy Java Simulations
○ Introduction ● Model ○ View
● Variables ○ Initialization ○ Evolution ○ Constraints ○ Custom
Main Variables
Add a new page
Copy this page
Move this page to the left
Move this page to the right
Rename this page
Enable/Disable this page
Show/Hide this page
Remove this page

Nan	Type	Dimension
time	double	
deltaTime	double	
amplitude	double	
frequency1	double	
frequency2	double	
phaseDelay	double	
x	double	
y	double	
	double	

Select the entry labeled *Add a new page*, and follow the familiar procedure of giving the new page a name; use, for instance, *Auxiliary Variables*. In this new page create the variables *maximum*, *minimum* and *n* with values *amplitude*1.2*, *-amplitude*1.2* and *150*, respectively.

Yes, you can use a variable in the value field for another variable, as long the former is defined before being used (exactly like we did with amplitude and maximum, for instance).

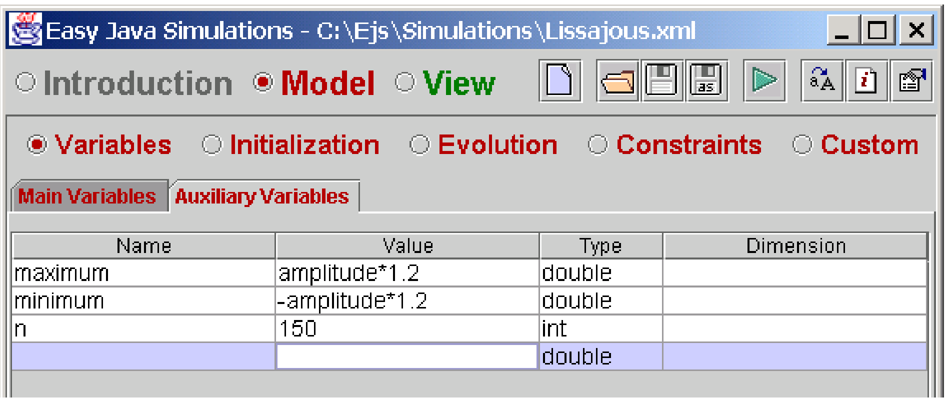
The variable *n* will only have integer values, we will therefore define it of integer type. For this, click on the field of the column *Type* next to the name of both variables, and a panel with the possible choices will open for you.



Name	Value	Type	Dimension
maximum	amplitude*1.2	double	
minimum	-amplitude*1.2	double	
n	150	double	

Select the *int* entry.

The second table of variables looks finally like this.



Easy Java Simulations - C:\Ejs\Simulations\Lissajous.xml

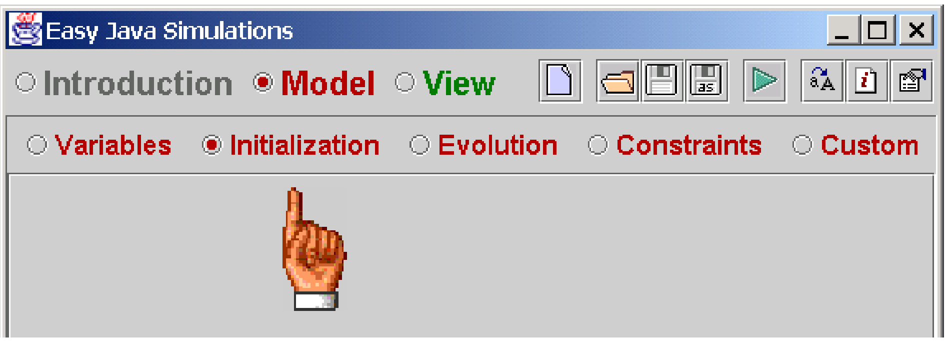
○ Introduction ● Model ○ View

● Variables ○ Initialization ○ Evolution ○ Constraints ○ Custom

Name	Value	Type	Dimension
maximum	amplitude*1.2	double	
minimum	-amplitude*1.2	double	
n	150	int	
		double	

3.4.2 Initializing the model

We will need no further initialization for our simple model, since we gave initial values to all variables. In some occasions, however, the model requires doing some computations to initialize all the variables. In these cases, we would use the *Initialization* subpanel of the *Model* panel. But as we said, we will not use it here and will leave this subpanel empty.



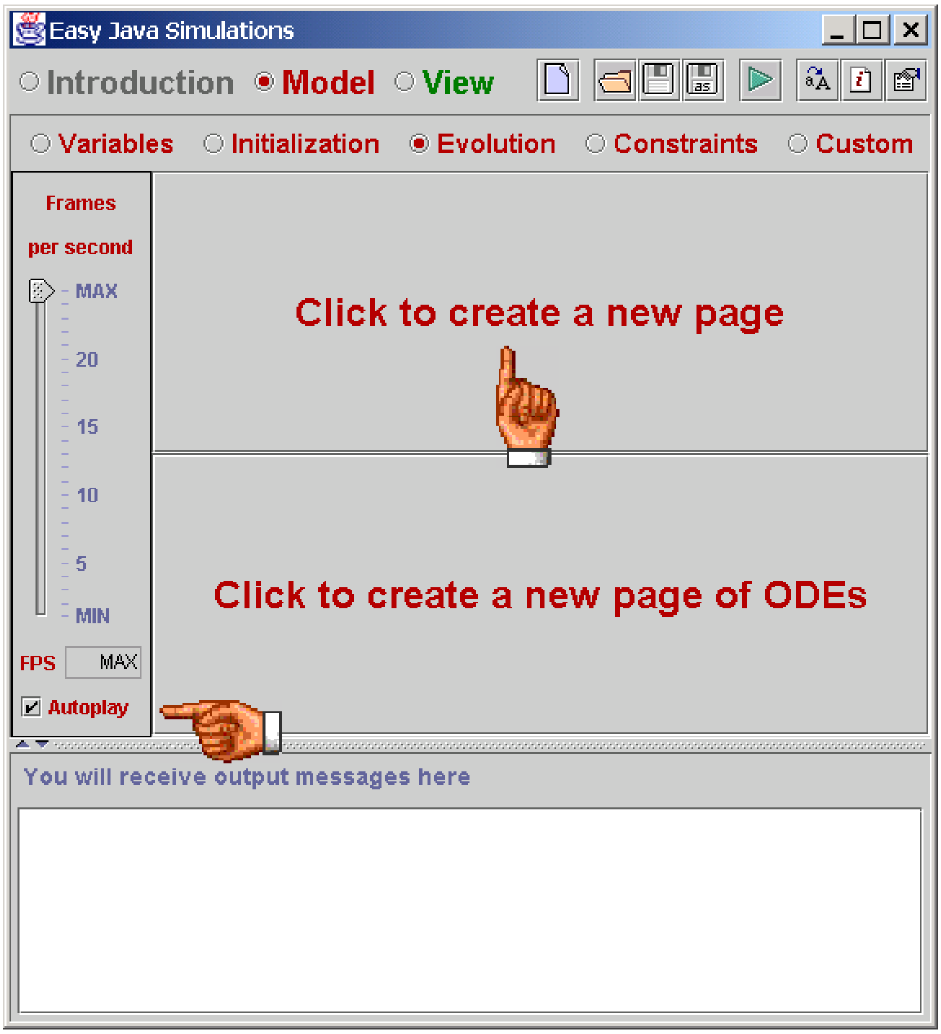
Easy Java Simulations

○ Introduction ● Model ○ View

○ Variables ● Initialization ○ Evolution ○ Constraints ○ Custom

3.4.3 Writing evolution equations

These equations define how the system evolves as time passes. The *Evolution* panel has a more elaborated interface.



Easy Java Simulations
○ Introduction ● Model ○ View
○ Variables ○ Initialization ● Evolution ○ Constraints ○ Custom
Frames
per second
MAX
20
15
10
5
MIN
FPS MAX
☑ Autoplay
Click to create a new page
Click to create a new page of ODEs

You will receive output messages here

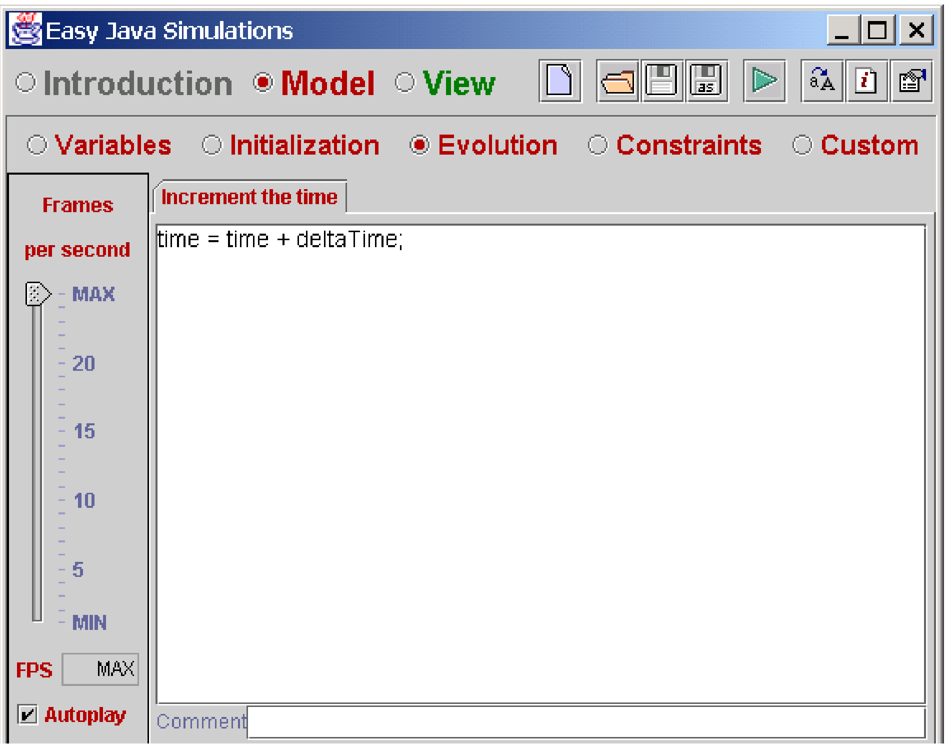
First of all, make sure that the *Autoplay* check box (the one the lower hand is pointing to) is activated (with a tick, as in the picture). This instructs the simulation to start as soon as it is run. If it were not, click on the button to activate it.

Now, click on the upper *Click to create…* area (the upper!) to create a blank new page and give it the name *Increment the time*. In this new page, write the equations that make the system evolve.

In our case, this reduces to the sentence:

```
time = time + deltaTime;
```

If you think that our model deserves some more lines of code, I need to ask you please to be patient, you will see the rest of the model when we write our constraint equations. Right now, the panel looks like this.


Easy Java Simulations
○ Introduction ● Model ○ View
○ Variables ○ Initialization ● Evolution ○ Constraints ○ Custom
Frames
per second
MAX
20
15
10
5
MIN
FPS MAX
☑ Autoplay
Increment the time
Comment

```
time = time + deltaTime;
```

3.4.4 Writing constraint equations

If you were wondering where should we write the equations that really define the model, that is,

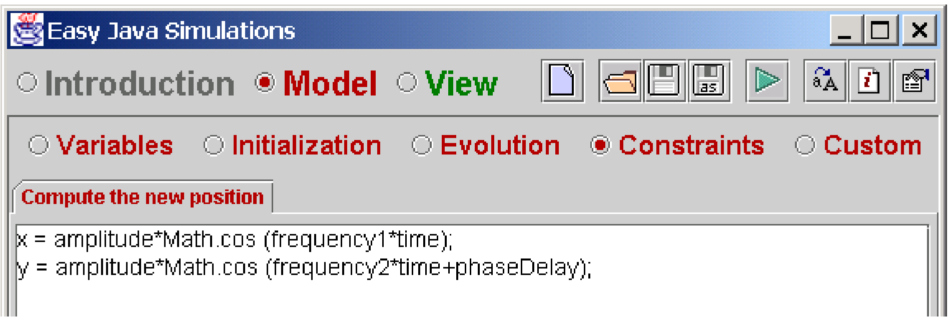
$$x = A_1 \cos \omega_1 t$$
$$y = A_2 \cos(\omega_2 t + \delta)$$

we just got to the point. It is a mistake (a common one, I must add) to write them as part of the evolution, because they are really something different. The difference is that, while *time* only changes when the system evolves, *x* and *y* are always modified according to these constraint equations no matter how and why any of the other variables change. This distinction is of special importance if we let the user change any of the parameters by direct interaction (something that, sooner ot later, we will want to do in an interactive simulation).

So, go to the *Constraints* panel, click on the *Click to*…area, give the new page the name *Compute the new position* and type the equations, which, in Java algorithmic dialect turns into the following,

```
x = amplitude*Math.cos (frequency1*time);
y = amplitude*Math.cos (frequency2*time+phaseDelay);
```

Here is the result.


Easy Java Simulations
○ Introduction ● Model ○ View
○ Variables ○ Initialization ○ Evolution ● Constraints ○ Custom
Compute the new position

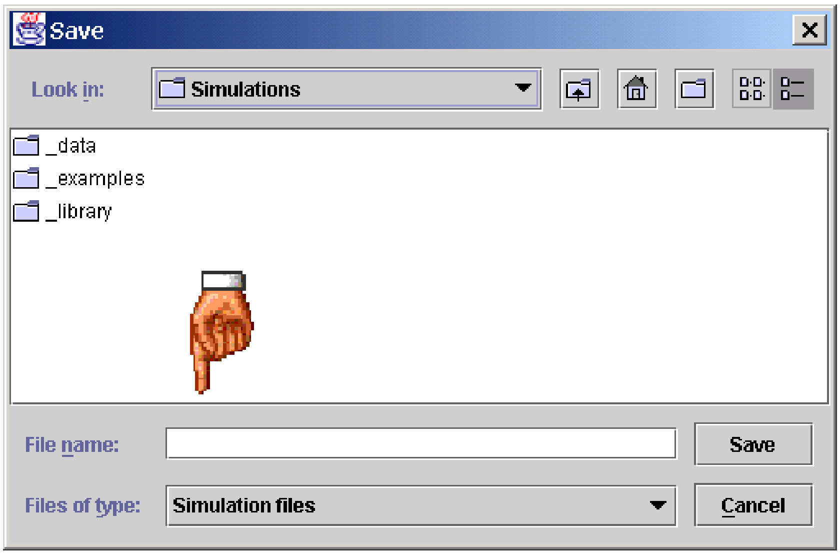
```
x = amplitude*Math.cos (frequency1*time);
y = amplitude*Math.cos (frequency2*time+phaseDelay);
```

3.4.5 Saving our work

Now, we have finished writing our model! At least in a first stage. The *Custom* panel is used when we want to define functions (methods, they are called in Java language) that can be reused in any other part of the simulation, or (very important!) to define model actions. But we will come back to this in a minute.

This is a perfect moment to save what we have done so far. It is not that we have finished our simulation or that we must save before going on. It is just that computers have the bad habit to sometimes stop, or as people usually say *hang*, without obvious reason and without previous notice. Also, not to blame only the computer industry, electric power could go off. Whoever is to blame, we don't want to loose our work (your work!), so we will just save from time to time.

For this, click on the 💾 icon. When you do it, you will be prompted with a file dialog like this



Look in: Simulations

File name:

Files of type: Simulation files

It is possible, if you started working in this chapter right after testing the installation, that this dialog places you in the _*examples* directory. In this case, click on the *Home* 🏠 icon. This will bring you back to your *Simulations* directory.

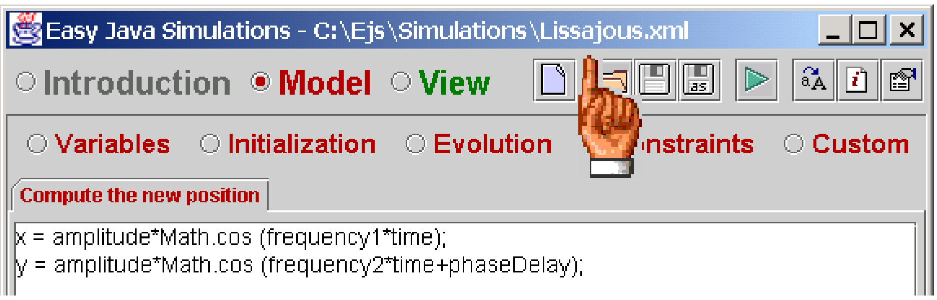
You need to provide the name of the file where you want to save your work in the *File name* field and then click on the *Save* button. **Ejs** assumes that you use the extension *xml* for this kind of files and will automatically append this extension, hence write simply *Lissajous* and click *Save*.

> Next time you click on the save icon, **Ejs** will use the same name, so you will not be prompted for a name again.

Ejs produces a reassuring message at the bottom output window that reads:

```
File saved successfully : C:\Ejs\Simulations\Lissajous.xml
```

You will also notice that the top banner of **Ejs** interface now includes the name of the file.



```
x = amplitude*Math.cos (frequency1*time);
y = amplitude*Math.cos (frequency2*time+phaseDelay);
```

3.5 Creating the view

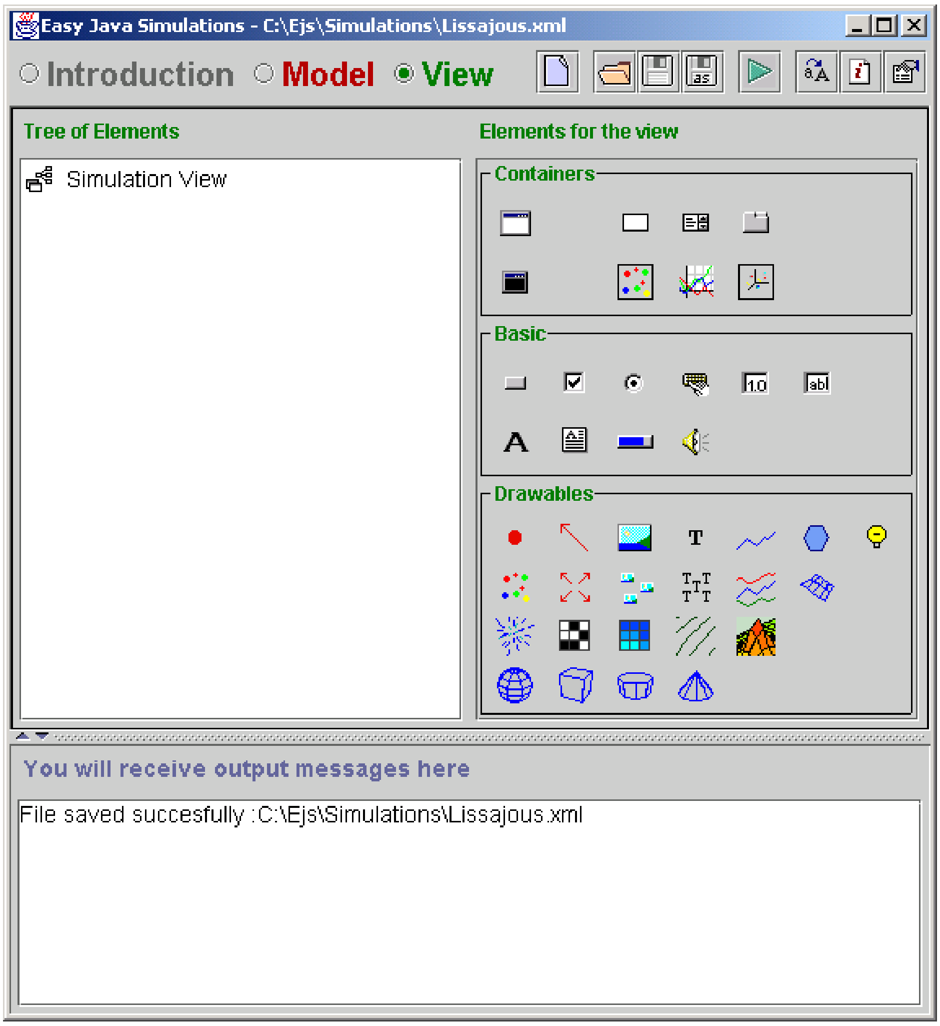
This part is going to be a bit longer and more complex if you are not familiar with computer graphics. I'll try to explain things in detail.

In **Ejs**, a view is created by adding graphical elements that work like building blocks which altogether form the interface of the simulation. Because there are elements that can host other elements, the result is a tree-like structure.

Usually, the first branch on this tree is a window element which is the one that will appear on the computer screen when we run the simulation and that will contain all the other elements of the view[14]. This window will also be the one that will appear on an html page if we run the simulation as an applet (see section 4.2).

When we select the *View* panel, the green one, (please do it now) we will see the interface for this panel.

[14] There can also be multi-windowed views, of course. Some of the sample simulations have views with two windows, actually. But let us keep our example simple, for the moment.


Easy Java Simulations - C:\Ejs\Simulations\Lissajous.xml
○ Introduction ○ Model ● View
Tree of Elements
Simulation View
Elements for the view
Containers
Basic
Drawables
You will receive output messages here

File saved succesfully :C:\Ejs\Simulations\Lissajous.xml

The central area contains two vertical subpanels. The left one is labeled *Tree of Elements* and the right one *Elements for the view*.

The panel with the title *Tree of Elements* displays exactly this, the tree-like structure of elements of the view, and is used to select and edit a particular element. It initially displays the root of the empty *Simulation View*.

The right panel works displays a column of three sets of icons, labeled *Containers, Basic* and *Drawables*, respectively. These sets group by functionality the elements that you can use to build the view for your simulation**.**

> Again, the number of elements that you will actually find in your copy of **Ejs** may be different from what is shown here. The reason is that new elements are continuosly been created and added to the standard distribution. Alternatively, **Ejs** may have been configured to display less elements than are actually available, for simplicity.

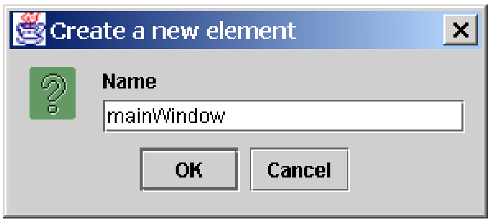
Although the description of how these components are selected and used will be given in section 6.4 , it follows a click and drop scheme, so it should be easy (say, not too difficult) to grasp the operation.

3.5.1 The oscilloscope's display

We will create a view that works similarly to an oscilloscope display. To begin with, we need an initial window in which we will place all other elements. For this first window we choose an element of type *Frame*. So, please, click on the first icon □ in the set of *Containers* at the top (if you place the mouse over it and wait for a second, a small caption will appear that reads '*A top level window*').

When you click on this icon, its background will change color and the cursor will change to a magic wand. Then, move to the *Tree of Elements* panel and click (with the magic wand) on the *Simulation View* node of the tree.

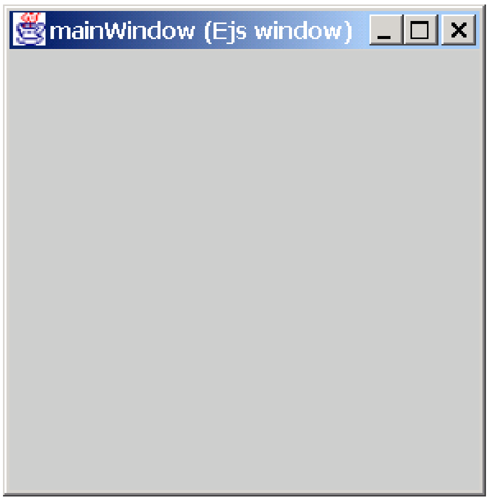
Ejs understands the trick and is about to create an element of type *Frame* as the first branch of the tree. But before doing so, it asks you the name you want to give to this element.



Create a new element

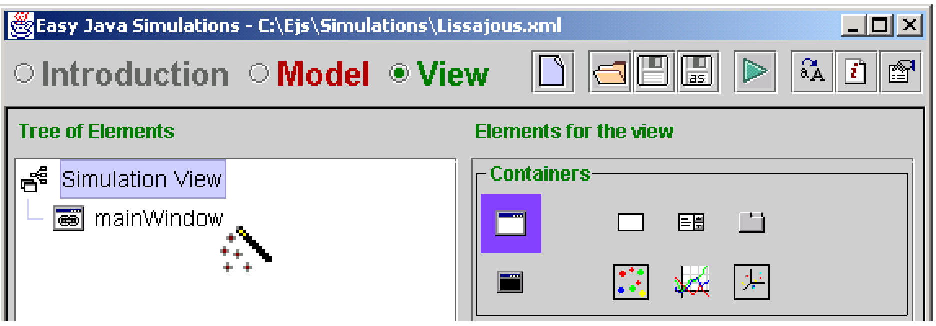
Name: mainWindow

Type in *mainWindow* and click *Ok*. You will notice that a new empty window appears on the screen (the size may vary from what is displayed here)



mainWindow (Ejs window)

and that the *Tree of Elements* reflects the change, too.



Easy Java Simulations - C:\Ejs\Simulations\Lissajous.xml

○ Introduction ○ Model ● View

Tree of Elements

- Simulation View
  - mainWindow

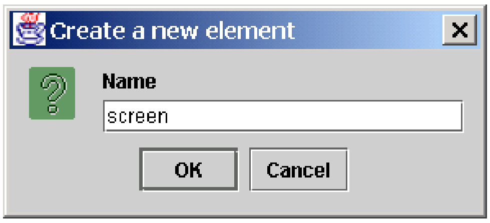
Elements for the view

Containers

Notice that the title area of the new window includes the name of the new frame, but it also specifies that this is a window created by **Ejs**. The text that appears on this title can be customized as we will see, but you will always be able to distinguish between a window created by Ejs and the corresponding window of a running simulation because the latter does not display the '*(Ejs window)*' suffix.

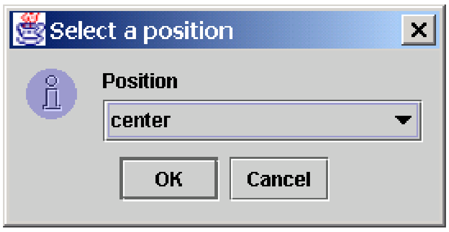
Now, we want to add to this frame an element that is able to contain display elements. The one we need is of the type *DrawingPanel*, has the icon and the caption that appears over it reads '*A 2D container for drawables*'.

Click on it, to make it the active element, then click with the magic wand on the recently created element *mainWindow* (either on the branch of the tree with this name or in the real window), and give the name *screen* to the new element that will be created.



Name: screen

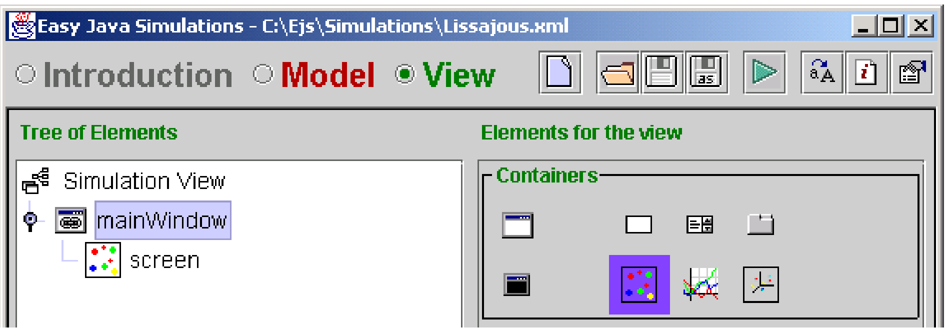
Now, something new happens! Before the element is actually created, we are prompted with the following dialog that asks us where to place the new element. This requires a bit of explanation.



Position: center

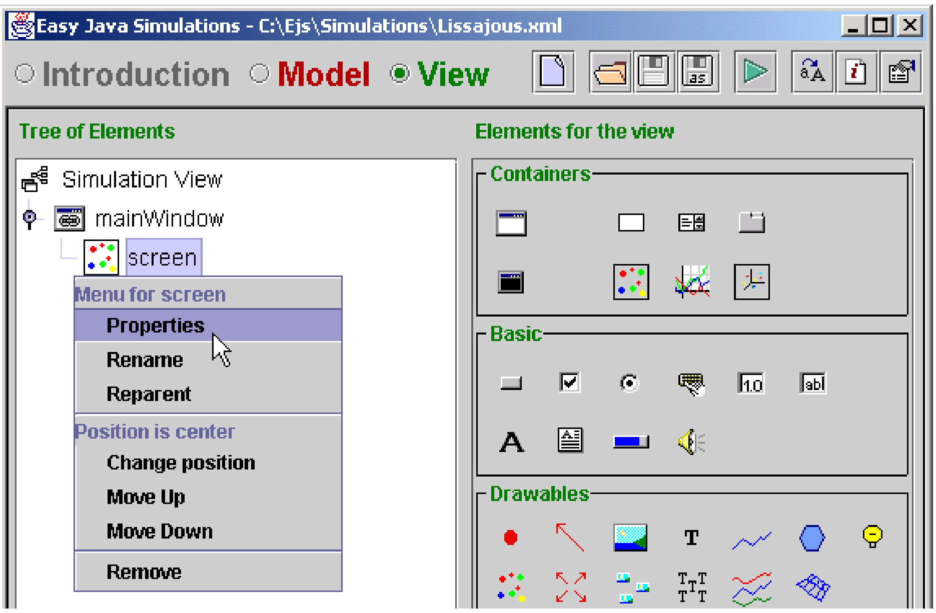
The frame element we created, *mainWindow*, is a container element which (by default) can hold up to five children elements, each at one of the positions it calls *north*, *south*, *east*, *west* and *center* - you can easily guess where it will place each of them. Our *screen* element will be right at the center, so accept the *center* option and click *Ok*.

Now, the tree of elements reflects the presence of a new element as a subbranch of *mainWindow*,

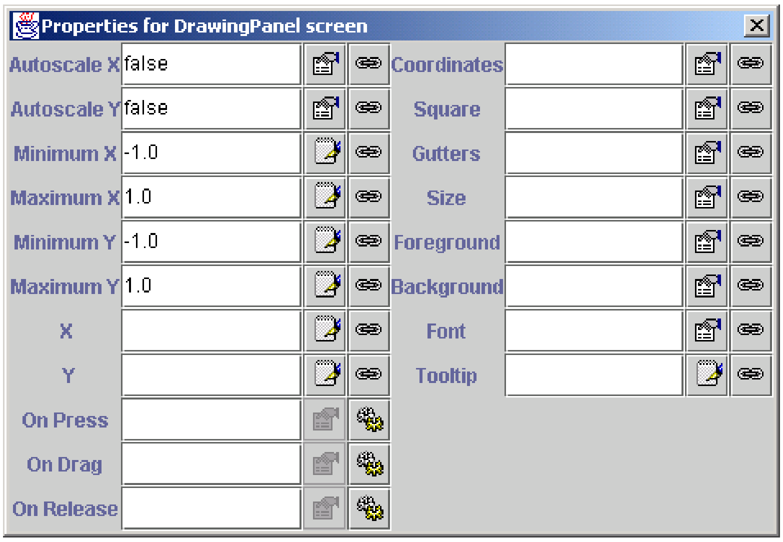


and *mainWindow* seems to change its background color to a particular type of blue. In fact, what has this new color is not *mainWindow*, but the newly created element *screen*.

This background color is the default for elements of the same type as our *screen* (drawing panels). But this can be customized. If we right-click on any element of the *Tree of Elements*, a popup menu will appear.



If we select the first option, *Properties*, a new dialog will show.

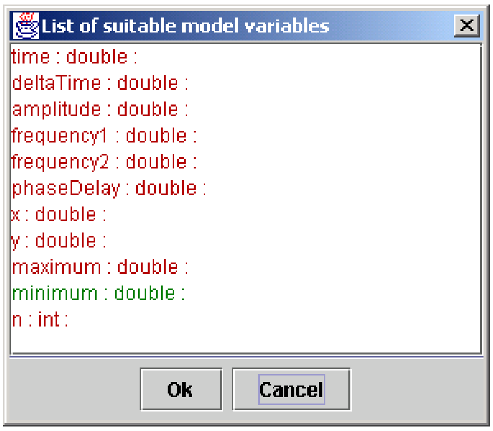


Autoscale X	false	Coordinates	
Autoscale Y	false	Square	
Minimum X	-1.0	Gutters	
Maximum X	1.0	Size	
Minimum Y	-1.0	Foreground	
Maximum Y	1.0	Background	
X		Font	
Y		Tooltip	
On Press			
On Drag			
On Release			

This corresponds to the edition window of this element. Each element behaves in a predetermined way which is basically given by the type of element that it is (that is, all frames behave in a manner typical to frames, buttons behave like buttons, etc.) However, elements have a set of properties that can be changed to customize it to better fit our needs.

All edition dialogs look pretty much the same. They only change in the number and particular properties for which they offer customization for the element. Part of the task of mastering the art of building views consists in getting familiarized with what types of element exist, what can they do for you and what properties can be edited. Besides learning by looking at the examples provided with **Ejs**, there is no other way than going to the reference page for every element in the appendices to learn about a given element.

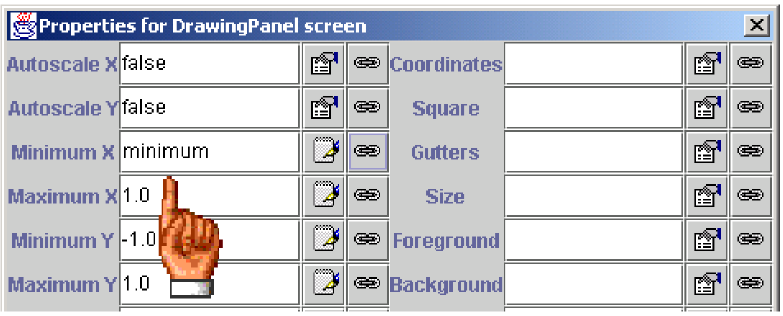
I will guide you in this task throughout this chapter. What we want to do is to go to the field labeled *Minimum X,* click on the icon 🔗 to its right, select *minimum* from the list of model variables that appear, and click *Ok*.



List of suitable model variables

```
time : double :
deltaTime : double :
amplitude : double :
frequency1 : double :
frequency2 : double :
phaseDelay : double :
x : double :
y : double :
maximum : double :
minimum : double :
n : int :
```

The edition dialog will reflect the choice in the corresponding text field.



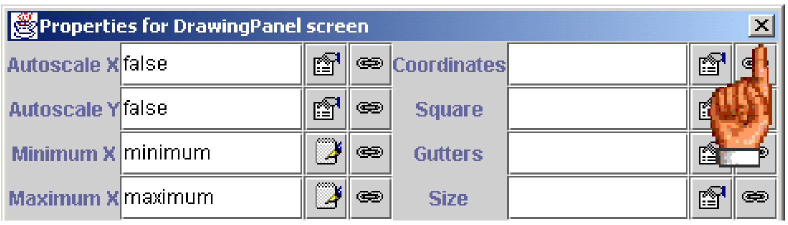
Properties for DrawingPanel screen

Autoscale X	false	Coordinates	
Autoscale Y	false	Square	
Minimum X	minimum	Gutters	
Maximum X	1.0	Size	
Minimum Y	-1.0	Foreground	
Maximum Y	1.0	Background	

An alternative way of achieving the same result would have been to type directly *minimum* in this field. But this is also more prone to typing mistakes.

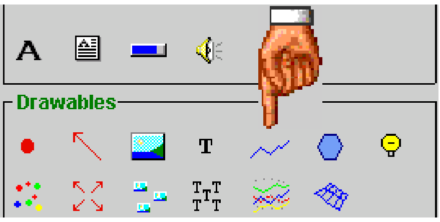
Now, repeat the procedure to give the value *minimum* to the property *MinimumY* and the value *maximum* to the properties *Maximum X* and *Maximum Y*. What we are doing is *linking* the properties of our *screen* that determine the region of the plane where it can draw, to the model's variables *minimum* and *maximum* (which we computed to make sure that Lissajous's figures would fit in their range).

This is all the customization we need from our drawing panel *screen*. You can close the edition dialog for it (by clicking on the 'close' button at the upper-right corner of the window).



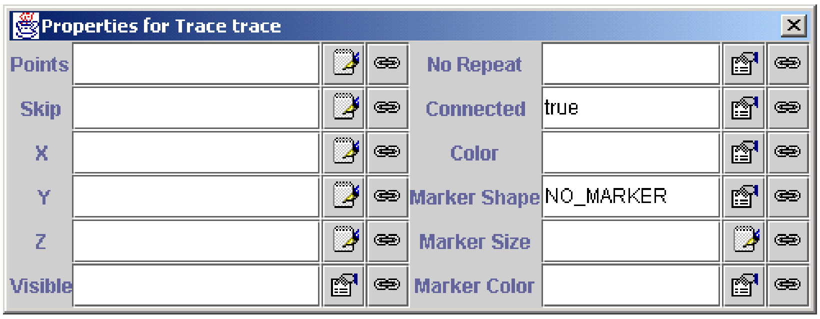
Properties for DrawingPanel screen

Autoscale X	false	Coordinates	
Autoscale Y	false	Square	
Minimum X	minimum	Gutters	
Maximum X	maximum	Size	



Now we need to actually draw on our *screen*. For this, we pick a display element (situated on the lower set of elements) of the type *Trace*, the one with the icon 〰. Its caption reads *A sequence of points*.

Add one of these, with the name *trace*, to the drawing panel *screen* (click on the icon to select it and then click with the magic wand on *screen*). Bring-in the edition dialog for the new element *trace* and on it,



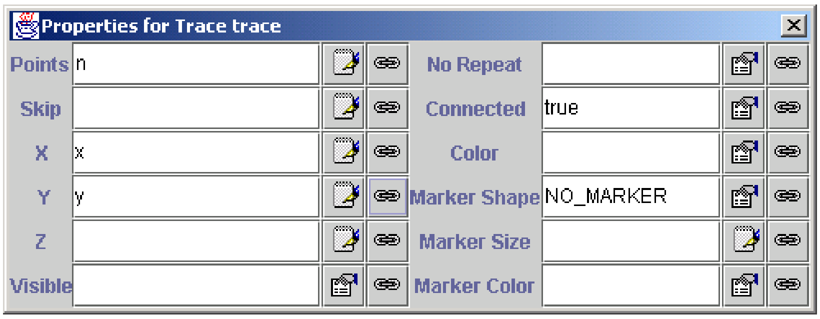
Properties for Trace trace

Points		No Repeat	
Skip		Connected	true
X		Color	
Y		Marker Shape	NO_MARKER
Z		Marker Size	
Visible		Marker Color	

link the following properties with the given model variables:

- property *Points* with variable *n*,
- property *X* with variable *x*,
- property *Y* with variable *y*,

and leave the rest as it was. The result looks like this.



Properties for Trace trace

Points	n	No Repeat	
Skip		Connected	true
X	x	Color	
Y	y	Marker Shape	NO_MARKER
Z		Marker Size	
Visible		Marker Color	

This setting means that this element will add a new point *(x,y)* (the *z* coordinate is unnecesary for our purposes and can be left empty) to the data set every time the evolution advances one step, up to a maximum of *n* entries. After this, it will discard an old point every time it adds a new one. In other words, the memory of this trace will be able to remember a maximum of *n* points.

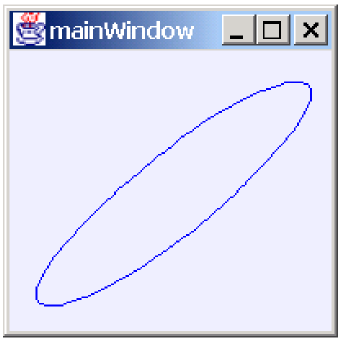
And this is the end of our view, at least in a very first stage. We will have to come back to it later, when we want to add some interactivity, but this is a good moment to see something running!

> A methodological remark. You may be thinking that, since *n* equals to *150*, we could have saved the effort of declaring a variable and all this, and could have just used the value *150*, instead.
>
> This is true, but it is also a short-sighted approach, since later on we may prefer to change this value to *200*, for instance, if we discover that more complex Lissajous' figures require more time to leave the proper trace (which is true, by the way). If we followed this second approach we would need to look all through the simulation to find where we used the value and then change the *150* to *200*, which is not only tedious but prone to error. Specially if we used the value in more than one place.
>
> Using our original approach we only need to change the value of *n*, which can be even done in run-time by the final user (if we provide her with a way of doing so, of course).

3.6 Running the simulation

We have therefore created the introduction, the model and the view for our simulation.

We can now run it. This is done using the icon ▶. Click on it and this is what you should see, but in motion.



If this is the case, congratulations, you completed your first simulation successfully!

3.6.1 Gallery of horrors (debugging)

It might well be, gentle reader, that at this very moment you are a bit frustrated because you didn't get this window at all! What happened, I would bet my kingdom on this, is that you made a minor typing mistake when copying what I asked you to type all along the chapter and now the compiler is complaining a bit, or most likely, a lot.

Do not worry very much about how many and how strange these error messages are. Feel yourself welcomed to a club with many members and be ready to look and find what your first error was. Many often, one single typing error causes a lot of error messages in cascade. It is therefore important to go to the top of the message area (at the lower part of **Ejs** interface), look for the very first error and try to correct it.

It is impossible for me to guess what was your error and give a solution for it, but instead, I can provide some examples of typical errors.

TYPICAL ERROR NUMBER 1.

Imagine that you typed

```
time = time + delta Time;
```

instead of

```
time = time + deltaTime;
```

when writing the code for the evolution page in section 3.4.3. See the difference? Yes, the space between *delta* and *Time*. If you did so and tried to run, you got the following error message:

```
Generating simulation file C:\Ejs\Simulations\Lissajous.java ...
C:\Ejs\Simulations\Lissajous.java:65: Invalid type expression.
   time = time + delta Time;  // > Model.Evolution.Increment the time:1
        ^
1 error
Compilation produced an error!
```

More important than understanding what the message *Invalid type expression*, means, is to be able to locate where the error is produced. Then, you can look at this place and try to figure out what you did wrong. In this case, **Ejs** is telling you that the error was produced in

```
// > Model.Evolution.Increment the time:1.
```

That is (in compressed jargon), line number 1 of the (model's evolution) page called *Increment the time*. Now, you have to recall that this is the name we gave to the evolution page. Go there, fix the error (delete the extra space) and try to run again.

TYPICAL ERROR NUMBER 2.

This one consists in believing that the case is not important in the name of the variables… It is.

If, in the same place as before, you typed

```
time = time + deltatime;
```

instead of

```
time = time + deltaTime;
```

You got the error message:

```
Generating simulation file C:\Ejs\Simulations\Lissajous.java ...
C:\Ejs\Simulations\Lissajous.java:65: Undefined variable: deltatime
   time = time + deltatime;  // > Model.Evolution.Increment the time:1
```

```
                ^
1 error
Compilation produced an error!
```

This time, the description of the error is a bit clearer (you defined a variable called *deltaTime*, not *deltatime*) and the location follows the same pattern as above.

TYPICAL ERROR NUMBER 3.

Imagine that you forgot to type the semicolon at the end of one line of the constraint page. That is, you typed

```
x = amplitude*Math.cos (frequency1*time)
y = amplitude*Math.cos (frequency2*time+phaseDelay);
```

instead of

```
x = amplitude*Math.cos (frequency1*time);
y = amplitude*Math.cos (frequency2*time+phaseDelay);
```

Just one semicolon produces the strange reaction:

```
Generating simulation file C:\Ejs\Simulations\Lissajous.java ...
C:\Ejs\Simulations\Lissajous.java:69: Invalid type expression.
   x = amplitude*Math.cos (frequency1*time)  // > Model.Constraints.Compute
the new position:1
   ^
1 error
Compilation produced an error!
```

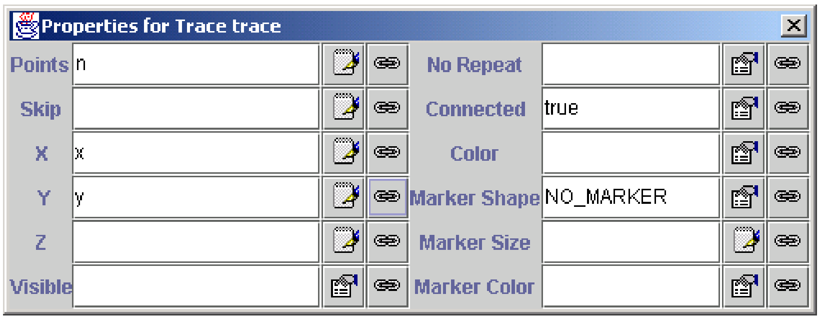
Again, ignore the error messages and look where it is placed. Go to line 1 of *Compute the new position* in (*Model.Constraints*) and type the semicolon.

Sometimes, error messages appear in cascade. A simple one produces a cascade of several others. For this reason, once more, my recommendation is that you try to correct the mistakes one after the other, starting with the first one, and trying to run after each correction.

TWO SPECIAL TYPICAL ERRORS.

A first typical error, which I have come across more than once, is to create the model and the view correctly, but to make a mistake while connecting them, or even forget to link them at all.

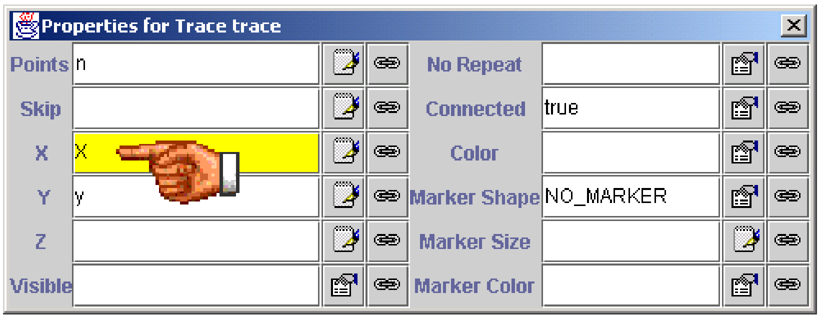
If, for instance, you forget to establish the links



Properties for Trace trace

Points	n	No Repeat	
Skip		Connected	true
X	x	Color	
Y	y	Marker Shape	NO_MARKER
Z		Marker Size	
Visible		Marker Color	

or if you typed, incorrectly



Properties for Trace trace

Points	n	No Repeat	
Skip		Connected	true
X	X	Color	
Y	y	Marker Shape	NO_MARKER
Z		Marker Size	
Visible		Marker Color	

(again, a difference in the case) you will not get any error message, but you won't obtain anything in the simulation, either.

Do not think the simulation is doing nothing. It is working hard running the model, increasing *time* by *deltaTime* and computing the *x* and *y* values as fast as it can. But we 'forgot' to tell it to move the beam to the new position, or used an incorrect link!

For good or for bad, the view can have its own variables (which don't even need to be declared beforehand), so it will assume that X is a new variable and since the model doesn't modify it, it will display nothing.

To avoid this type of error, I strongly recommend using the pick-up button to the right of the input field, whenever possible.

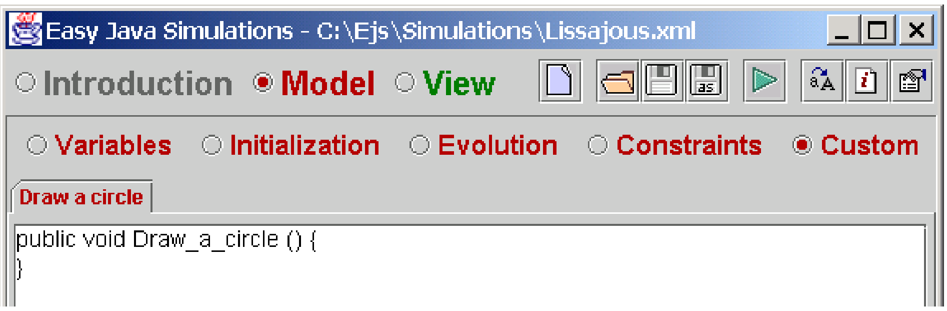
The second typical error is to forget to check the *Autoplay* check box on the *Evolution* panel of the model. Then, the simulation is correct but, this time, it is actually doing nothing because it has not started to run. You would click on ▷ and get a static, disappointing simulation.

3.7 Adding interactivity

3.7.1 Defining custom actions

As we said, we will define some actions that will help us set the value of the variables that produce nice Lissajous' figures. For this, we go back to the main *Model* panel and make use of its *Custom* subpanel, which you have to select now. The interface we obtain is very similar to the one you are familiar with, but with a small difference.

Click on the *Click to ...* area to create a new empty page and give it the name *Draw a circle*. You will see that (in contrast with what happened in previous panels) the text area doesn't appear empty.



```
public void Draw_a_circle () {
}
```

The reason is that pages in the *Custom* panel allow us to create *any* kind of Java method, and this requires us to specify what we want (which in turn forces us to know a little bit about how Java methods are defined). We shall not get into details here, but this basically means that we need to give it a name, a return type and declare its accessibility. Optionally, we can declare the method to accept input parameters.

In our case, the accessibility is going to be kept *public*, the return type *void* (so we accept the proposed values for both)*,* the name we change to *setCircle* (the proposed name[15] is too long for my taste) and we leave the parameter list empty *( )* as it is now.

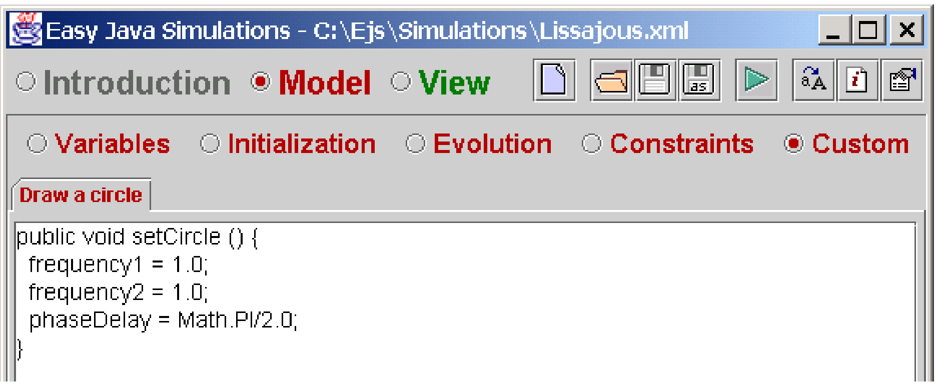
We now type in the body of the method (between the curly braces *{* and *}*) the following sentences, which produce a circular Lissajous figure,

```
frequency1 = 1.0;
frequency2 = 1.0;
phaseDelay = Math.PI/2.0;
```

(Math.PI is Java's special way of referring to the constant number *π=3.1415…*).

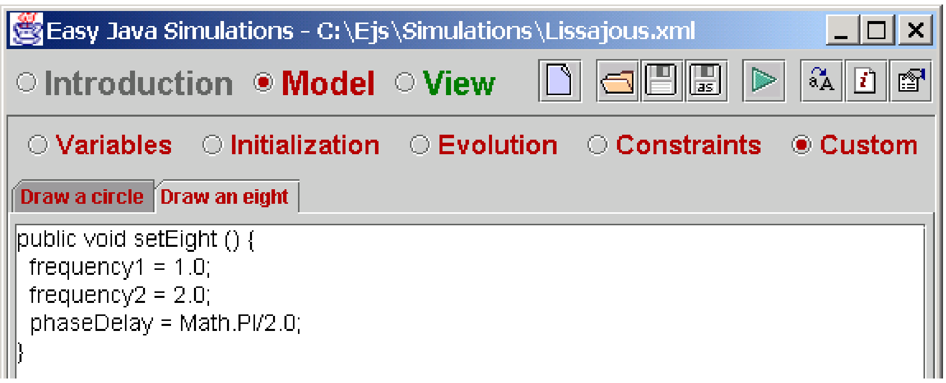
[15] which matches the name of the page – with spaces replaced by underscores, since method names can not have spaces.

The result is


Easy Java Simulations - C:\Ejs\Simulations\Lissajous.xml
○ Introduction ● Model ○ View
○ Variables ○ Initialization ○ Evolution ○ Constraints ● Custom
Draw a circle

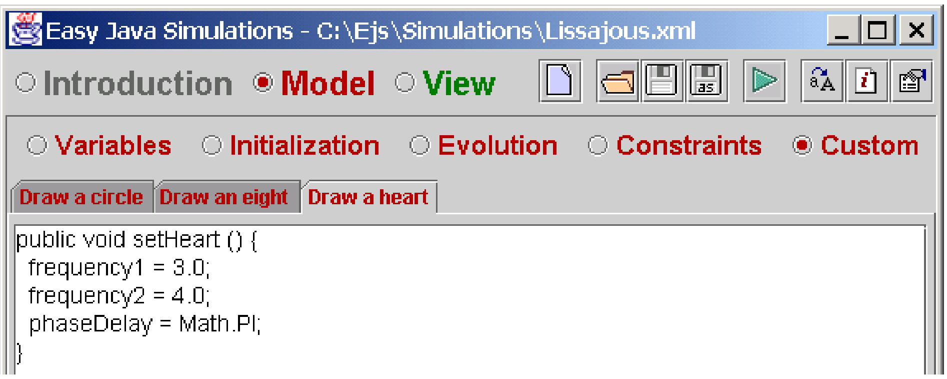
```
public void setCircle () {
 frequency1 = 1.0;
 frequency2 = 1.0;
 phaseDelay = Math.PI/2.0;
}
```

This custom method is ready. Now, create two more with the following names and codes.


Easy Java Simulations - C:\Ejs\Simulations\Lissajous.xml
○ Introduction ● Model ○ View
○ Variables ○ Initialization ○ Evolution ○ Constraints ● Custom
Draw a circle
Draw an eight

```
public void setEight () {
 frequency1 = 1.0;
 frequency2 = 2.0;
 phaseDelay = Math.PI/2.0;
}
```

This one gives a figure which looks like a horizontal 8.


Easy Java Simulations - C:\Ejs\Simulations\Lissajous.xml
○ Introduction ● Model ○ View
○ Variables ○ Initialization ○ Evolution ○ Constraints ● Custom
Draw a circle
Draw an eight
Draw a heart

```
public void setHeart () {
 frequency1 = 3.0;
 frequency2 = 4.0;
 phaseDelay = Math.PI;
}
```

And this one a figure which looks (more or less) like a heart.

These are all the actions that we need. And this also finishes our model. Again, this is a good moment to save our work (click on the 💾 icon).

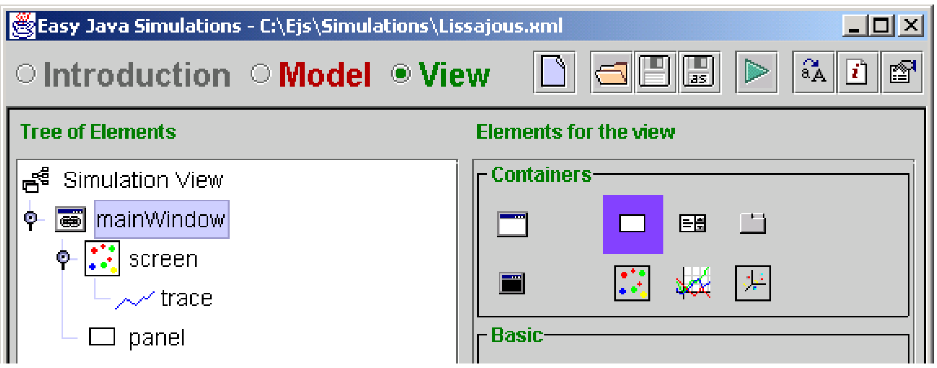
3.7.2 Adding interface buttons for our actions

To finish our simulation, we will now create three buttons on the simulation's view that will help us trigger the actions just defined. For this we will first create a container panel that will hold them.

Go back now to the *View* main panel. The element we are interested in is the set of *Containers* and has the icon □. If you place the mouse over it and wait for a second, a small caption will appear that reads '*A basic container panel*'.

Click on it to select it and hit with the magic wand the *mainWindow* entry of the *Tree of Components* window. This time, accept the proposed name *panel* for the new element and select *west* as the position for this panel.

Ejs now reflects, in the *Tree of Components* window, the component we have just created, that is, *panel*.



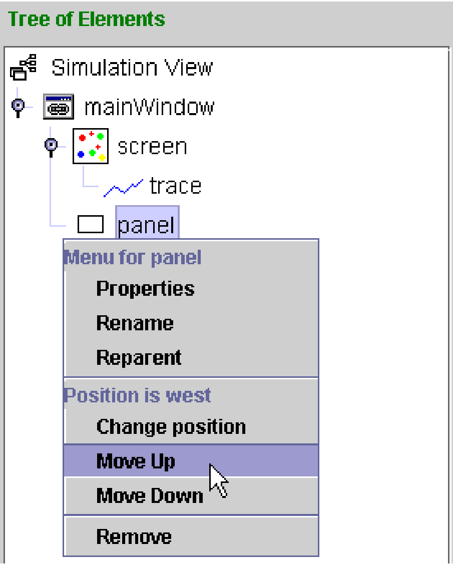
However, our display, the oscilloscope's window, will not have changed at all. This is because, although the new panel is there, it is still empty, so it doesn't ask *mainWindow* for space.

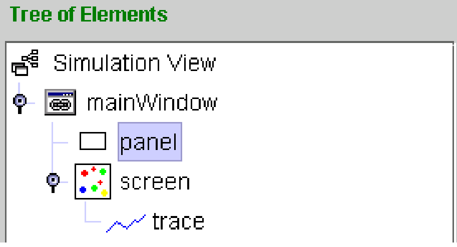
Before proceeding, just for organizational reasons, although we have created *panel* later that *screen*, we want it to be the first child of our *mainWindow*.

> There is actually no real reason for doing this (in this case), but this lets me show you the feature of reorganizing the hierarchy of children ☺.

For this, bring-in *panel*'s popup menu and click on *Move up*.

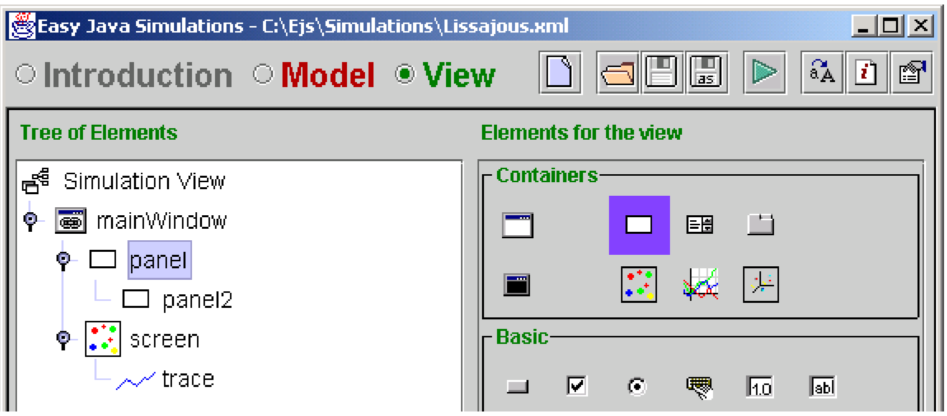
As a result of this action, both *panel* and *screen* will have switched position.





Now, we will create a second panel as child of *panel*. Since the panel element is already selected, click with the magic wand on *panel*, accept the proposed name *panel2* and select *north* as position for it.

You have now created a second panel and the *Tree of Components* looks like this.



Easy Java Simulations - C:\Ejs\Simulations\Lissajous.xml

○ Introduction ○ Model ● View

Tree of Elements

- Simulation View
  - mainWindow
    - panel
      - panel2
    - screen
      - trace

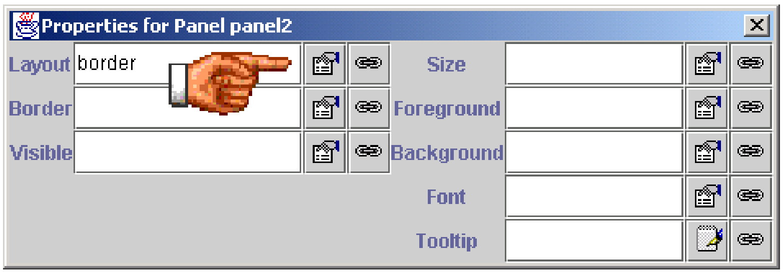
Elements for the view

Containers

Basic

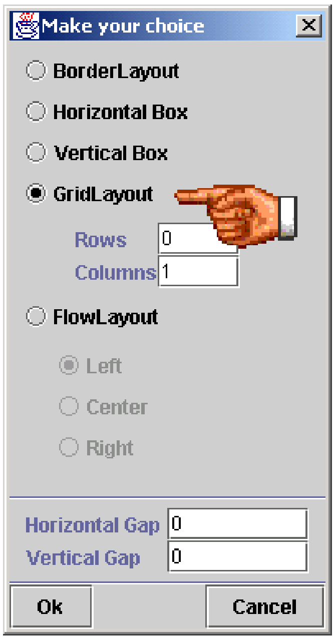
However, the oscilloscope still looks exactly the same. To make it finally change, we will create three buttons on *panel2*. However, we will first change the way the parent *panel2* will accommodate the new children.

Bring in the edition panel for the properties of *panel2* (right-click on *panel2* and select *Properties*).



Properties for Panel panel2

Layout	border	Size	
Border		Foreground	
Visible		Background	
		Font	
		Tooltip	



Make your choice

○ BorderLayout

○ Horizontal Box

○ Vertical Box

● GridLayout

Rows 0

Columns 1

○ FlowLayout

● Left

○ Center

○ Right

Horizontal Gap 0

Vertical Gap 0

Ok Cancel

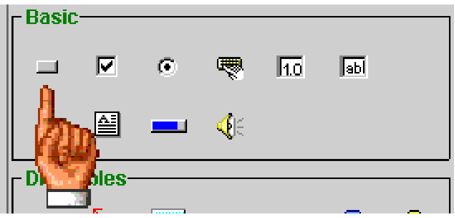
Click on the 📝 icon to the right of the text field for the property labeled *Layout* (which reads now *border*) and you will obtain a specialized dialog.

This dialog lets you choose the layout policy for the container panel. This is the policy, or scheme, that the container will use to allocate space for its children.

In it, choose the option labeled *GridLayout* (the one the hand is pointing to) and click *Ok*. This corresponds to a grid with (as the default says) one single column.

Now, we can add the three buttons we need.

In the panel with the menu of *Elements for the view*, select the element with the following icon ▬ , the caption would read *A button for actions*.

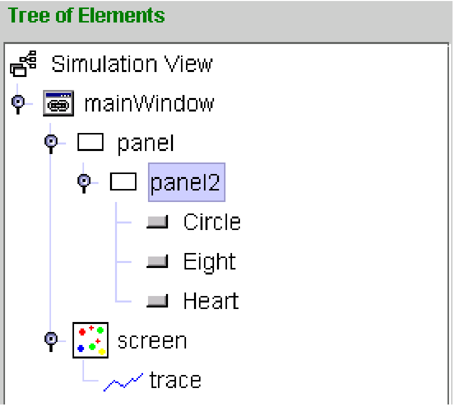


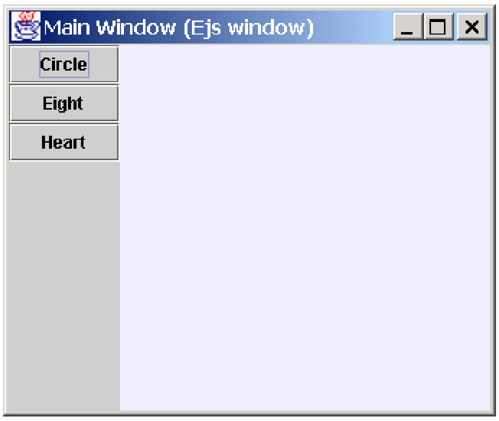
Click on it and be ready to click on *panel2* three times to create three buttons.

You will notice that this time, *panel2* does not ask you for a position for the new children, but that the new buttons are just created (well, after giving them a name; use the names *Circle*, *Eight* and *Heart*). This is because its layout imposes the position of any new child according to its 'order of birth'.

Please create the three buttons now.

Now, the oscilloscope display has changed (but see next note!) Here are both the *Tree of Components* and the display that you should see.





If the panel with the buttons doesn't show up, you may need to resize imperceptibly our *mainWindow*. Sometimes, only when a container gets resized does it assign its children the space they need.

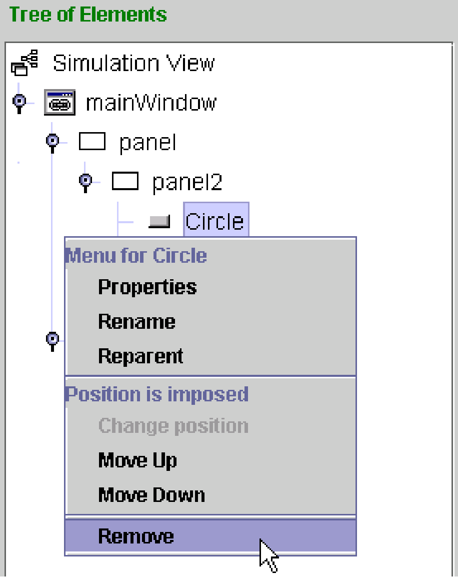
To resize the window, do as with any other window on your screen, click on any of the corners and drag the mouse a little bit while holding the mouse button down.

STILL ANOTHER TYPICAL ERROR.

Perhaps, but only perhaps, you clicked in the wrong window and got a button where you did not want it to be. For instance, imagine that you clicked on *panel* when you wanted to click on *panel2*.

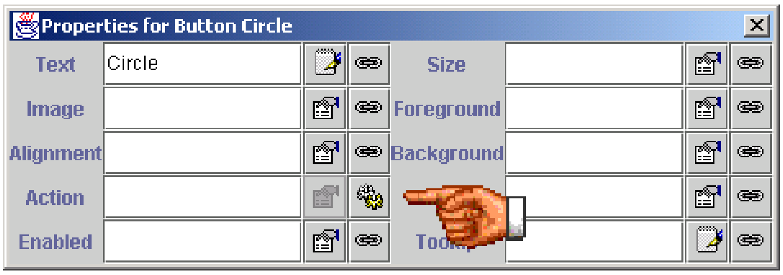
In this case, the best thing to do is to remove the newly created button and create it again.

To remove an existing component, bring in the element popup menu in the *Tree of Components,* and use the last entry in this menu. The picture reflects this menu for the first button.

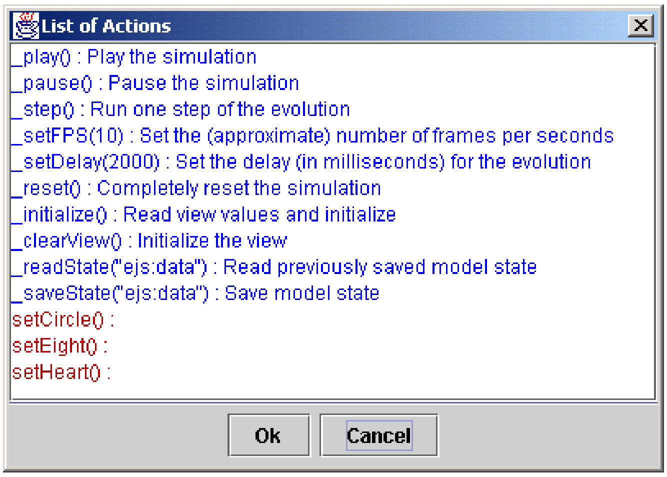


To make our view look better, you may need to resize our *mainWindow* to make the *screen* element a bit bigger and more or less square sized.

Finally, we will associate to the buttons the corresponding actions. We do this using the edition dialog for each of the buttons. Right-click on each of them and click on the edit button to the right of the input field of the property labeled *Action*.



Next figure shows the window that appears with the list of available actions.



_play() : Play the simulation
_pause() : Pause the simulation
_step() : Run one step of the evolution
_setFPS(10) : Set the (approximate) number of frames per seconds
_setDelay(2000) : Set the delay (in milliseconds) for the evolution
_reset() : Completely reset the simulation
_initialize() : Read view values and initialize
_clearView() : Initialize the view
_readState("ejs:data") : Read previously saved model state
_saveState("ejs:data") : Save model state
setCircle() :
setEight() :
setHeart() :

The actions written in blue color correspond to some predefined actions, such as starting -*_play()*- or stopping -*_pause()*- the simulation. In red (the color for model items) we find the actions we defined in our model.

> The parentheses indicate that our action will correspond to a Java method in the generated Java file. Although this does not matter to us, it is important that you respect the parentheses.

Select *setCircle()* for the button *Circle*, *setEight()* for the button *Eight* and *setHeart()* for the button *Heart*.

This means that when the simulation runs, if the user clicks on any of the buttons (say in button *Circle*), the code we wrote in the corresponding action (in *setCircle( )*) will be executed. This is exactly what we want.

Let us summarize what we have done in this section. We have extended the view to provide three more buttons. For this, we first created some panels that we needed to place the buttons in the right place.

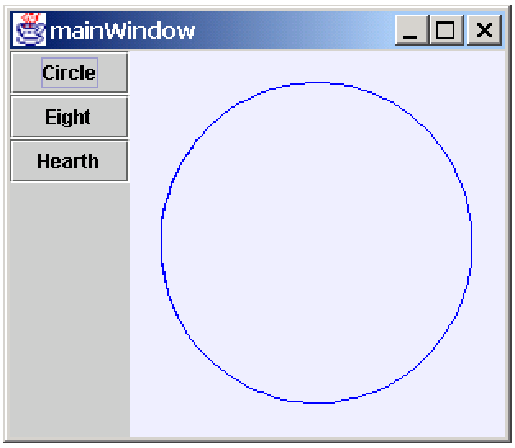
These button display meaningful labels (since they use by default their name and we gave them meaningful names) and we have associated our control actions to them, so that when we click on them (while running the simulation) they will execute the code we wrote in the corresponding actions.

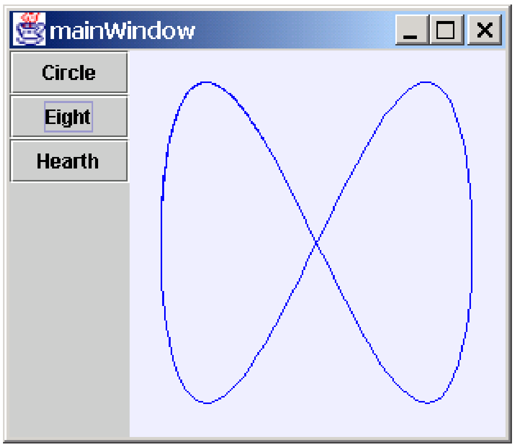
Our simulation is finished and we are ready to run it!

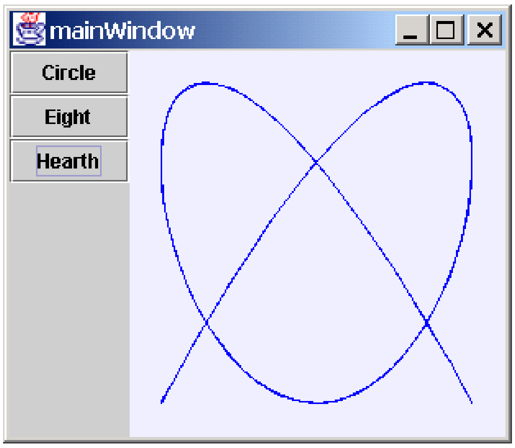
3.8 Running the complete simulation

Run the simulation again, by clicking on the ▶ icon and you get your complete simulation!

If you click on the three buttons you'll get the following Lissajous' figures:







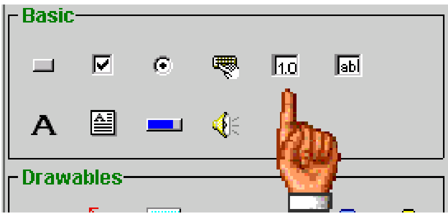
3.9 Adding some more interactivity

Now that we have finished duty, let us continue a little bit, just for the joy of it! Or, if you prefer being more serious, take this as the exercise section of this chapter.

We want to add to our simulation the possibility to directly enter the values of *frequency1*, *frequency2* and *phaseDelay*, which will let the user produce all possible Lissajous' figures.

For this, and assuming you already gained some familiarity with **Ejs** interface, go to the *Containers* set of the *View* panel and create a new container panel under *panel* (make sure you select *panel* and not *panel2*), using *south* as position. Give the new panel a one-column grid *Layout* (the same as *panel2*),

In this new panel, which is called, most likely, *panel3*, create now three elements of the class *NumberField* which belongs to the group *Basic*, whose icon is 1.0 . Accept the proposed names for them.



Now, edit the property called *Format* of each of the new NumberField elements to read *Freq1 = 0.0*, *Freq2 = 0.0* and *Phase = 0.00*, respectively. These *0.0* and *0.00* are not values, but a way of formatting numerical values.

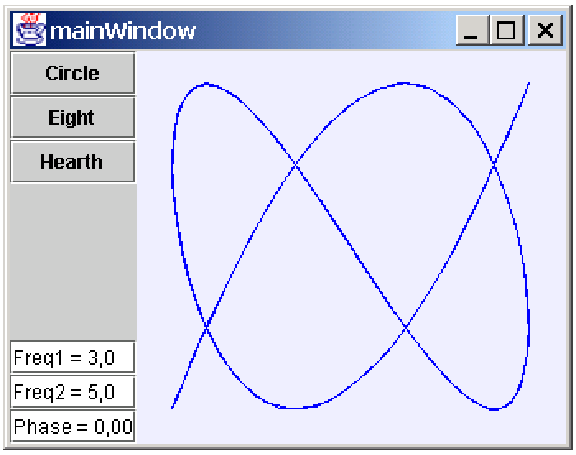
> To learn more about this, you will have to read the reference page for *NumberField*, in the appendices of this manual.

For each of the fields, link also the property called *Variable* to *frequency1*, *frequency2* and *phaseDelay*, respectively.

Similarly to the links we established before, these new links mean that each NumberField element will display the actual value of the corresponding variable. But also that, conversely, if we type a new value in the field, the model variable will accept it as its new value.

If you now run the simulation (don't forget to save it, too, just in case), you will read the values of the three variables, *frequency1*, *frequency2* and *phaseDelay*, in the corresponding fields and you will also be able to enter a new value by clicking of the *Field* elements, typing a new value and hitting enter.

For instance, if you use the values *3.0*, *5.0* and *0.0*, you will obtain the figure to the right.



Freq1 = 3,0
Freq2 = 5,0
Phase = 0,00

Finally, you can also play with values of frequencies that are not integers, and you will get curves with change in time. Try, for instance, the values *1.005*, *2.0* and *0.79*.

But obviously, I can not reproduce changing curves here. Now you need a real computer to continue exploring Lissajous' figures!

This page intentionally left blank

4

4. Using your simulation

Now that you have created your first simulation, you may be so fond of it that you actually want to use it in your classroom, or perhaps even publish it on the Internet! You'll find instructions on how to do this in this chapter.

Simulations created with **Ejs** can be run in three different forms. The first one is by using **Ejs** itself, as you have already done yourself. The second is by running the generated simulation as an *applet* using any of the most popular Web browsers. The third and final form is to run the generated simulation as an independent Java application.

The first option is the one that would make me happier, since it would help spread the use of **Ejs**. It has a very important advantage: your user can learn how you actually simulated a given phenomenon. This has, in my opinion, a great pedagogical added-value. It also has a disadvantage, however: the user needs to have **Ejs** installed in her computer and needs to know how to use it. Although you and I agree that **Ejs** is a wonderful tool, perhaps you don't want all your users to learn to use it just to be able to run a given simulation.

Fortunately, simulations created with **Ejs** are, once generated, independent of it. Hence, in this case, the second option is the one I usually recommend: running the simulation as a Java applet.

A Java applet is a special form of application that is designed to be run within a Web browser. The browser loads a type of file, called an *html* file[16], that includes a special tag (or instruction) which tells the browser to run the required applet within its own window. This html file may reside either in your local disk or in an internet location, served to you by a Web server.

I also recommend this option because the html file allows you to wrap the simulation with explanatory text that will introduce to the student the

[16] *html* stands for Hyper Text Markup Language.

phenomenon being simulated, and also lets you prepare a simplified control of the simulation from the Web page using JavaScript [17].

Although you already created a bit of html code when we wrote the *Introduction* part for our simulation (and **Ejs** will use this), you may want to create a more complete and sophisticated html page with the help of one of the popular html editors, which can help you achieve a much more professional result.

The third option, running the simulation as a stand-alone Java application, is only recommended when you want to run a simulation that writes data to your disk. As it is explained in part II, simulations created with **Ejs** can read data from the disk and also through the Internet, from a Web server, no matter in which of the three ways they are executed. However, in order to create this data, the simulation must write it to disk and, for security reasons, Java applets can not write to the local disk. In this case, you have to run the simulation as a Java application.

Distributing (i.e. giving to others) a simulation to be run within **Ejs** is straightforward, you only need to give your simulation (the *.xml*) file away. In our example, you would distribute the file called *Lissajous.xml*. Your user would need to know how to start **Ejs**, load this file in it and run it, as explained in section 2.4 .

> If your simulation needs some data to run properly, like an image file, for instance, you need to distribute this data too.

This chapter is devoted to explain how to run simulations in the latter two ways, as well as to instruct you how to distribute your simulations independently of **Ejs**, that is, how to give them to other people so that they can also run them even if they don't have or don't want to use **Ejs**.

This is a good moment to say that you are free to distribute as many copies as you want of any simulation you create with **Ejs** with no cost for you and no copyright limitation. Only if you distribute **Ejs** itself you need to accept **Ejs** license limitations, which, on the other hand, are very low demanding: just respect my name and copyright notices both in **Ejs** and in this manual. Obvious, isn't it? ☺.

[17] Describing JavaScript and html format is clearly out of the scope of this manual, we will very briefly describe what we need for our purposes. You can surely find an appropriate book on any of these subjects in your nearest technical library.

4.1 What happens when you run a simulation

I need to describe briefly what happens when you run a simulation with **Ejs**, just before the simulation view actually appears on the screen. I will illustrate the process using the example of our previous chapter.

If you gave, as suggested, the application the name *Lissajous,* you will find in your **Ejs** *Simulations* directory, after you run the simulation for the first time, the following files (if you don't find one or more of these files, don't worry very much and wait until you read next subsection):

- *Lissajous.xml*. This is the simulation file itself. It contains all the work we did in the previous chapter stored in a special format called *xml*. This is certainly the most precious file of all the family of *Lissajous* files in this directory. Do not delete unless you want to lose your work!

  > It is possible, and very acceptable, that you save this file in a different directory. This doesn't affect Ejs at all and might be even be convenient for organizational purposes, after you have created several dozen simulations ☺. In this case, this file won't appear in **Ejs** *Simulations* directory

- *Lissajous.java*. This is the generated Java source file. Recall that **Ejs** falls into the category of code-generators: it takes all the pieces of information that you provided through its interface and builds a Java program out of it.

  There is a long way to go from one place to the other, but this is taken care of *automagically* by **Ejs**. This generated Java file uncovers some of my programming secrets, and might be of use for you if you want to learn how I did it, or if you are an experienced Java programmer and want to modify it directly. If you are not interested in it, you can also instruct **Ejs** to remove this file when it doesn't need it any more. See next subsection for this.

- *lissajous.jar*. This is the output of the Java compiler when it processes the file *Lissajous.java*. Well,… not really. Compilation can produce a lot of small files. What I have done is to instruct **Ejs** to put everything in one single file and then compress it, so that to facilitate distribution. This file is the final result of this process.

- *Lissajous.html*. This is a sample html file that will help you run your simulation as an applet, and is the central piece of next section. Depending on the configuration of your options for **Ejs** (again, read next subsection, please), and how many pages you wrote in the *Introduction* part of your simulation, you may also find more hml files for this simulation, all of them will start with the prefix *Lissajous*.

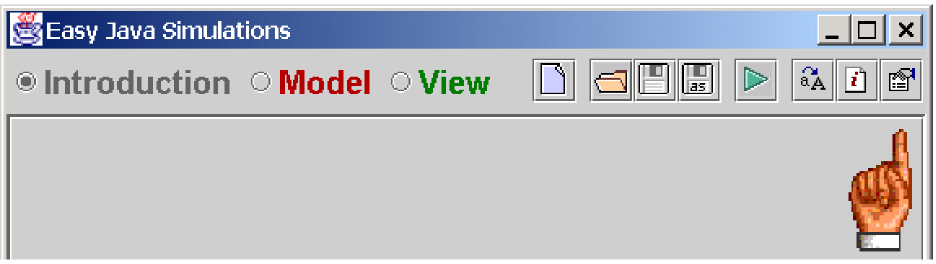
- *Lissajous.bat*. Finally, this is the file that you need to run your simulation as an independent Java application and will be discussed in section 4.3 .

You must also find the directory *_library* in this directory and, probably, there are other files in this directory created as result of running some sample simulations.

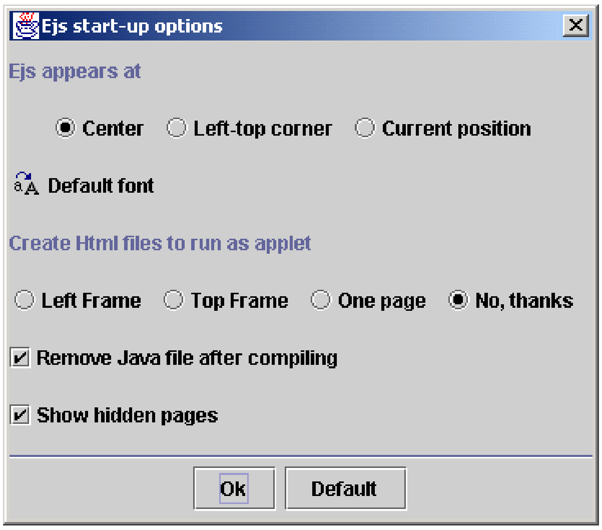
> Recall, as instructed in section 2.3 , that you can only use as **Ejs** home directory a directory that holds a copy of the directory *_library*. This directory is also needed when you want to distribute your simulation about Lissajous's figures, as described in section 4.4 .

4.1.1 Ejs configuration options

If you look at your *Simulations* directory and can't find some of these files, this is not necessarily due to a malfunction in your copy of **Ejs** . **Ejs** has some configuration options that can be modified to instruct it not to generate files that you are not going to need. To check these options, click on the options icon which you will find in the right-most upper row of icons:



You will get the options editor, you can see to the right. In it, you see the options that **Ejs** offers for customization, exactly with the values that they had the last time **Ejs** was run. The options you see in the picture are, in fact, the options I use myself.


Ejs start-up options

Ejs appears at

● Center ○ Left-top corner ○ Current position

Default font

Create Html files to run as applet

○ Left Frame ○ Top Frame ○ One page ● No, thanks

☑ Remove Java file after compiling

☑ Show hidden pages

You can decide where **Ejs** main window should appear: at the center of your screen, at the upper-left corner, or wherever it was the last time you used it.

This is useful if you prefer **Ejs** to appear always at a given position. In this case, just place **Ejs** main window there and select the *Current position* option.

You can also select the font **Ejs** will use as default every time it is run. This produces the same resut as changing the font for the current session (which is done using the font icon aA in the main toolbar,) if only this option has a more permanent effect: next time you run **Ejs** it will also use the font you select here.

Proceeding downwards, you can instruct **Ejs** how it should create html pages for your simulation,and whether it should keep the generated java file or not. This certainly affects what was described in the previous section.

Ejs can generate html files either following closely the way you structured your *Introduction* part for the simulation (that is, each introduction page in a separate html file), or putting everything in one single file. You can even tell **Ejs** to generate no html at all! I use this option myself when I am going to use and distribute my simulations as **Ejs** files. This way, my *Simulations* directory doesn't get crowded by html files I am not actually going to use. For the same reason, I also use the *Remove Java file…*option.

Finally, the last option offers you to display (or not) hidden pages. As we will learn in part II of this manual, hidden pages are pages that form part of a given simulation, but that the creator of it, prefers you not too see.

> Or at least, not to see them at a first stage, since, of course, by checking this option you can always see this page, whether the author wants it or not.

This is useful when you, as author, are preparing simulations for your students and you want them to concentrate their attention in the relevant parts of the simulation, ignoring other parts that are either routinary or that don't contribute too much to the core of the task. Hence, you could hide these other parts.

4.2 Running your simulation as an applet

Let's go back to the main purpose of this chapter.

As said above, the best way of using your generated simulation is to run it as a Java applet by opening an html page that includes it.

Whenever you successfully run a simulation using **Ejs**, a sample such page is created for you. You can find this file (perhaps together with other auxiliary html pages) in your working directory. It has exactly the same name of your simulation, but has a different suffix, *.html*.

You can run the simulation just by loading this file (load precisely this one, not any of the other auxiliary html files) into your favorite Web browser. There is, however, an important remark. **Ejs** and the simulations it generates, use Java components of a special (and advanced) graphical family, called *Swing*. This family requires that your browser supports Java 2.

Unfortunately, not all browsers come *Java 2 enabled* 'out of the factory'. Some require that you install an update of what is called Java *plug-in*. The latest version I have tested thoroughly is 1.3.1 and you can download it at Sun's site under the link http://java.sun.com/getjava/download.html.

> I have also tested version 1.4 and it works fine most of the times.Usually, it even runs faster. Unfortunately it also produces 'unexpected results' (not to call it *bugs*) from time to time. I therefore recommend version 1.3.1.

4.2.1 What is in the html file

In this release, **Ejs** can generate a number of html pages that combine to provide a nice set of web pages for your simulation, according to the information we placed in the *Introduction* part of **Ejs**. From all of them, we describe here only the one that holds the simulation. This would be (for our *Lissajous* simulation) the file *LissajousSimulation.html*.

> However, if your configuration options instruct Ejs to put all the html code in one single file, you will see the code that follows forming part of the one and only *Lissajous.html*.

This file has three different parts, which I list separately.

The first part is a standard header that all html files need.

```
<html>
 <head>
  <title> Home page for Lissajous</title>
 </head>
 <body>
```

This declares the file as an html file, gives it a title and opens the section *<body>*, where the real content is.

The second part is the central piece of the page. It contains the tags for a separator and for the inclusion of the simulation within the page.

```
The simulation's view should appear right under this line.<br>
<applet code="org.colos.ejs.LauncherApplet.class"
        codebase="." archive="_library/_ejsLibrary.jar,lissajous.jar"
        name="Lissajous"  id="Lissajous"
        simulation="lissajous.Lissajous"
        capture="mainWindow" width="338" height="239">
</applet>
```

Finally, the last part includes the html code needed to include some JavaScript buttons that control the simulation and the closing of the page.

```
<!--- Finally the JavaScript buttons --->
<br><hr width="100%" size="2"><br>
<p>You can control it using JavaScript. For example, using buttons:</p>
<p>
<input type="BUTTON" value="Play"
```

```
          onclick="document.Lissajous._play();";>
    <input type="BUTTON" value="Pause"
          onclick="document.Lissajous._pause();";>
    <input type="BUTTON" value="Reset"
          onclick="document.Lissajous._reset();";>
    <input type="BUTTON" value="Step"
          onclick="document.Lissajous._step();";>
    <input type="BUTTON" value="Slow"
          onclick="document.Lissajous._setFPS(1);";>
    <input type="BUTTON" value="Fast"
          onclick="document.Lissajous._setFPS(10);";>
    <input type="BUTTON" value="Faster"
          onclick="document.Lissajous._setFPS(1000);";>
    </body>
    </html>
```

I have written in bold font above the words that depend on the actual generated simulation. In the example above, of course, this includes the word *Lissajous*, but also the name of our main window and its size.

The special applet tag is made of the lines

```
    <applet code="org.colos.ejs.LauncherApplet.class"
        codebase="." archive="_ library/_ejsLibrary.jar,lissajous.jar"
        name="Lissajous"  id="Lissajous"
        simulation="lissajous.Lissajous"
        capture="mainWindow" width="338" height="239">
    </applet>
```

It states that the browser should load the applet *LauncherApplet* which will, in turn try to run the simulation *lissajous.Lissajous* that we have created. This *LauncherApplet* is a utility class, included in the file *_ejsLibrary.jar* (which is in the *_library* directory) that handles all the internal tricks needed to run your simulation within an html page.

It also states that the applet should be created with a size of **338** times **239** pixels, because this is the size we used (to be more precise, that *I* used when writing this manual) for the main window of our simulation.

You see in the second line of this tag construction that the applet also needs to load the archive *_library/_ejsLibrary.jar* which must be in the *codebase* directory *'.'* (that is, the same directory in which the html file is located). This is the reason why the *_library* directory must be there!

The page that we created in the *Introduction* part of **Ejs** will be included in a separate html page. If we create more than one of these pages, then we will also get more html pages, each of which will appear in a frame created by the central html file, the one you should read in your Web browser, which is called *Lissajous.html*.

4.2.2 JavaScript control of the simulation

The final part of the html listed above shows how the simulation can be controlled using JavaScript commands.

JavaScript is a scripting language that is understood by most Web browsers and that allows basic programming within html pages. We shall not describe JavaScript here, but only mention that JavaScript can be used to perform actions on our simulation. This use of JavaScript provides in fact a second way of controlling the simulation besides the simulation interface itself.

JavaScript can be used to:

- Execute any of the predefined control routines: *_play*, *_pause*, etc. (see part II)
- Execute any of the public actions that we defined in our model
- Set the value of any of the variables of the model

The procedure for this is always the same. You have to include an input button like follows:

```
<input type="BUTTON" value="Pause"
        onclick="document.Lissajous._pause();";>
```

This one uses the *_pause( )* predefined method that will stop the simulation. Now, if you click on the button labeled *Pause* that will appear in your browser, the simulation will stop.

To use any other method, just replace the *_pause()* part with any other valid method of our simulation. For custom model methods, you need to specify the prefix *model*. For instance, if you want to execute the method called *setCircle* that we defined in the model part of our *Lissajous* simulation, substitute *_pause()* with *_model.setCircle()*[18].

If the method you want to call accepts variables, like *_setDelay (int delay)*, or *_readState(String filename)*, just write the parameters within the parentheses. However, methods that accept parameters of type *String*, must enclose them in inverted commas, instead of quotes. Like in

```
<input type="BUTTON" value="Nice Figure"
       onclick="document.Lissajous._setVariables
         ('frequency1=3.0; frequency2=5.0; phaseDelay=0.0');";>
```

By the way, *_setVariables* is a very useful predefined method of any **Ejs** generated simulation that can be used to set the value of any variable of it. In this example, it is used to set the value of *frequency1* to *3.0,* of *frequency2* to *5.0* and of *phaseDelay* to *0.0*. Of course, like in this example, your model must

[18] At the time of this writing, and for a reason I ignore, Internet Explorer doesn't seem to deal with this type of methods properly, while other browsers, like Netscape and Mozilla, do.

have variables with the corresponding names. Notice that sentences of the form *variable=value* are separated by semicolons.

4.2.3 Adding your own html text

If you know how to modify an html file, you can go ahead and personalize the sample html file according to your needs. For instance, by including a background image, or configuring the page with frames,…

4.3 Running your simulation as an application

The third way in which you can use your simulation is as a stand alone Java application.

However, you must notice that running Java programs requires a Java virtual machine present in your system. Browsers incorporate them (either by default or after installing a plug-in), but if you want to run the simulation as an application you must obtain and install Java run time environment. You can find it in Sun's site http://java.sun.com.

Notice, however, that if (according to section 2.2) you installed Java SDK in order to be able to run **Ejs**, then the run time environment is already present in your system.

If you want to run your simulation as an application, then the file we are interested in is *Lissajous.bat*. If we inspect it[19], we find the following simple content.

```
@echo off
c:\jdk1.3\jre\bin\java -classpath "_library/_ejsLibrary.jar;lissajous.jar"   lissajous.Lissajous
```

The first line is not so important, it is just instructing the operating system to work in silent mode. That is, it should not repeat next line, just execute it.

The second line is the important one. It is calling Java run-time engine (which in my system is placed in the directory *c:\jdk1.3\jre\bin*) and telling it to execute the class lissajous.*Lissajous* which lives in the *lissajous.jar* archive file. It also tells it to use (for its own internal purposes) the archive file *_libray/_ejsLibrary.jar*. Again, the *_library* directory must be in the same directory as the batch file for the simulation to run.

[19] We are illustrating the batch Windows file.

If you run this file[20], an empty operating system window would open and the simulation will run in a second window.

Also, if you know how your operating system works, you can edit this file to suit your needs, like placing the files in a different directory, start the operating system window in minimized mode, etc.

4.4 Distribution of simulations

As I said above, simulations created with **Ejs** are independent of it, if only they need the library that includes the graphic components you used when creating the view and other internal files necessary for operation.

These are not part of the standard Java graphic library but have been created, or adapted specifically to **Ejs** with the goal of making them useful to visualize scientific phenomena and data with a unified simple interface of use. Included in this library is also a subset of the *Open Source Physics Tools* created by Wolfgang Christian (see http://www.opensourcephysics.org).

However, distributing your application is very easy. You just provide the user with the generated files described in this chapter plus the library directory *_library*. If your application uses its own data (like an image file) you'll need to provide this data too.

If you are distributing your simulation using a CD-ROM, perhaps even using floppies, just copy the files corresponding to your application in the distribution media and add a copy of the *_library* directory, too.

If you are using a Web server to publish your simulations through a network, maybe the Internet, place the generated files in an accessible directory on the server and a copy of the *_library* directory in the same place as your simulations.

Of course, if you are an expert in publishing applets on the Internet, you can always improve this setup. But these simple instructions will work.

Once more, I need to recall that anyone using your simulations from the internet will require a Java 2 enabled browser, as specified in section 4.2 .

[20] In Windows just double-click on it.

PART II. Detailed description

This page intentionally left blank

5. Building models with Ejs

5.1 Definition of a model

We create the model of a phenomenon when we define what its relevant magnitudes are, set their values at an initial moment and establish the rules that govern how these magnitudes change.

We use the word magnitude to refer both to state variables (values that describe the phenomenon) and to parameters (values used by the governing rules). When we write our simulation, we will refer to magnitudes as *variables*.

A variable can hold a value that changes as the simulation runs or one that does not. In other words, it can represent a constant or variable magnitude; but this doesn't make much difference to us now[21].

Hence, the state of a model is given by the value of its variables $x_1, x_2, x_3, ..., x_n$.

> Usually, to help understand a model, variables are given meaningful names in concrete models (such as velocity, acceleration, number of individuals of a species, concentration of a chemical element, etc…). Since our exposition is general, we assume that they all have the same name with different indexes.

The state of a model at a given moment (the values of its variables) can change due to two reasons:

- the inner dynamics of the simulation, which we call evolution, and
- the influence of external agents; i. e. the direct interaction of the user of the simulation.

In the second case, it is possible that the user changes one or more variables that affect others which are dependent on them. We then say that there are *constraints* among these variables.

[21] It is possible that one magnitude, considered in principle constant, changes later; either because we change the laws of the model, or because the user interacts to directly modify the magnitude. Hence, the term variable turns out to be finally appropriated.

Both processes of change are ruled by equations that describe the laws under which the evolution takes place or the interdependence of the magnitudes, transcribing them into mathematical formulas.

Therefore, to specify the model of a simulation we need to set the initial state, the evolution equations, and the constraints equations.

5.1.1 The initial state of a model

First of all, we must define the variables of our model. In many cases, this is a crucial process from which a good or bad simulation can follow (choosing the right reference system, magnitudes that simplify the formulas…). Hence, this is our very first step to create a model:

Step 1 : Define the variables

When the variables of the model are chosen, setting the initial state consists in given them the right values.

Step 2 : Initialize the variables

5.1.2 Evolution and constraint equations

Using the terminology stated above, the system can evolve autonomously from the current state $x_1, x_2, x_3, ..., x_n$ to a new one $x_1^*, x_2^*, x_3^*, ..., x_n^*$, simulating the passing of time (which, by the way, can or cannot be one of our variables).

We call the equations ruling this transition, *evolution equations*, which can be written as a system of one or more equations of the form

$$x_i^* = f_i(x_1, x_2, ..., x_i, ..., x_n)$$

Step 3 : Write evolution equations

Sometimes these laws have a direct formulation, as in the case of discrete systems of the form $x_{n+1} = f(x_n)$. In occasions, they derive from the discretization of differential equations.

Simulating the evolution of a model in time consists in computing, from the current state of the model $x_1, x_2, x_3, ..., x_n$, the new values $x_1^*, x_2^*, x_3^*, ..., x_n^*$; take these as the new state of the model, and continuously repeat this process for as long as the simulation runs.

Besides the equations associated to the autonomous evolution of the system, we must also describe its reaction to changes forced by external agents. As said before, to changes imposed directly by the user.

The change of a given variable, caused by the user, may affect others that are related to it. We call these interrelationships, *constraints*, and are expressed by one or more equations of the form

$$x_i = g_i(x_1, x_2, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$$

where a variable can only appear at one side of the equation.

Step 4 : Write constraint equations

If, at any moment, a variable on the right-hand side of a constraint changes, the equation is evaluated and the variable on the left-hand side modified. Since this happens within the same instant of time, evolution equations are **not** evaluated.

It is possible to have models with only evolution equations, models with only constraints and models with both. In this last case, if an external agent changes a variable, only constraint equations are evaluated, but if a step of the simulation takes place, then evolution equations are evaluated and, immediately after, constraint equations.

When writing a model, it is convenient to clearly identify which equations correspond to evolution and which to constraints. A useful criterion (though perhaps not always valid) is to examine the equation we use to compute the new value of a variable. If this value depends on the current value of the same variable, then this is an evolution equation. If it doesn't, then it is a constraint.

A typical example is a model described by an ordinary differential equation of the form $x'(t)=f(t,x(t))$, for which we know a closed-form solution $x=sol(t)$. In this case, we may be tempted to write evolution equations in the form

```
t* = t + dt;
x* =sol(t+dt);
```

when the evolution equation really reduces to

```
t* = t + dt;
```

and the second is in fact a constraint:

```
x = sol(t);
```

The difference is significant, because if the user modifies the value of *t*, this immediately implies a change of the current value of *x*, even if no step of the evolution is taken. This would not happen if we wrote the equations in the first form.

We have a different situation if we do not know a closed-form solution of the differential equation. Then, we must try a solution based on a numerical method, and evolution equations take the form

```
t* = t + dt;
x*= computation(t,x,dt);
```

where the computational formula depends on the differential equation and on the numerical method we use. Now, if the user changes *t*, it is not possible to compute the new value of *x* without going through all intermediate evolution steps.

> Instead, we can interpret that what the user really wants is to set new initial conditions for the equations, changing $x(t_{current}) = x_{current}$ to $x(t_{new}) = x_{current}$.

5.1.3 Running the model

Once these four steps have been completed, we have finished creating the model. If we run the simulation,

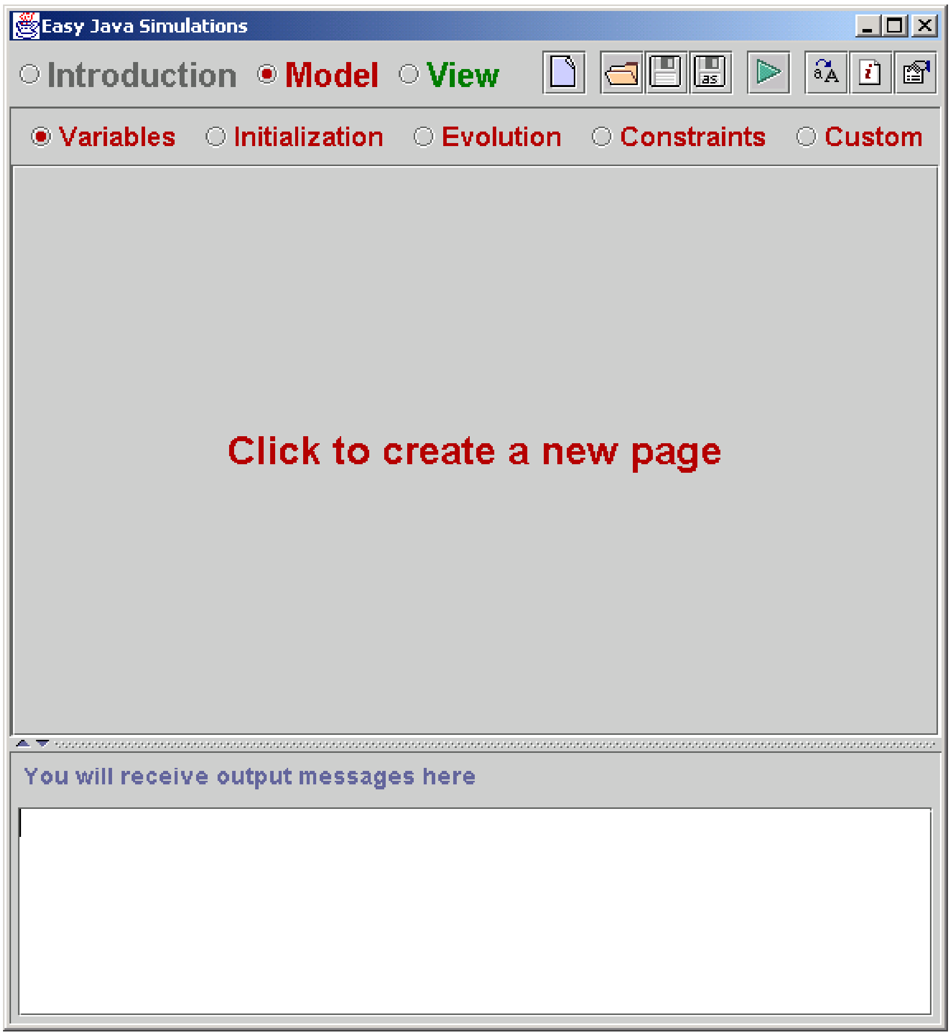
1. Variables are created and their values are set to those indicated in the initialization step.
2. Constraint equations are evaluated (since the initial value of some variables may depend on the initial values of other variables).

Now, the model is in its initial state and the simulation waits for an evolution step or for the user to interact with it.

3. In the first case, evolution equations are evaluated and, immediately after, constraint equations. We then reach the new state of the model in a new instant of time.
4. In the second case, when the user changes a variable, constraint equations are evaluated and we obtain a new state of the model within the same instant of time.

5.2 Ejs interface for the model

Corresponding to the four steps described above, the part of the interface of **Ejs** dedicated to the implementation of the model has four radio buttons, one for each step, and each with its own central panel.



Click to create a new page

You will receive output messages here

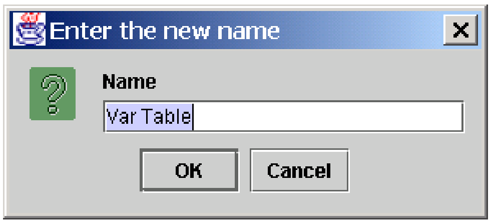
Yes, I know; you actually see five! But the last one, labeled *Custom*, is just a place for you to write custom Java methods (also called sometimes functions or subroutines) which can help you write more elegant programs. It is also used to create control actions for the simulation as we will describe in section 5.7 .

The interface for the panels corresponding to each of the radio buttons is rather similar, only the one for the evolution is a bit more complex.

They all show initially an empty area with a message inviting you to create a new page for variables or code.

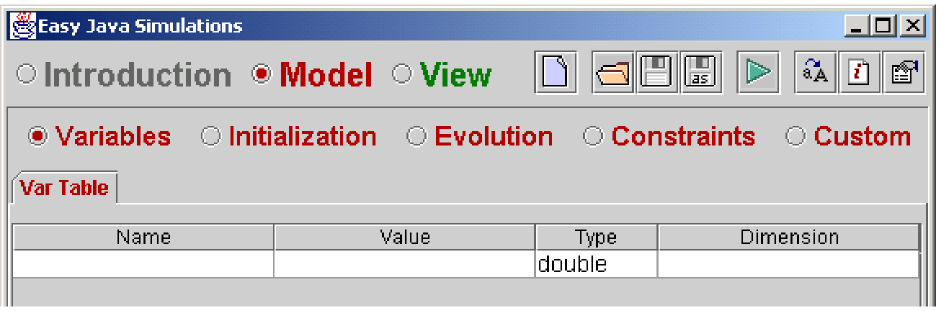
If you click on this panel, you are prompted to provide a name for this page.

Though **Ejs** proposes you a generic name for the new page, I recommend that you use descriptive names, so that anyone who reads your simulation can figure out what this page contains.



Name: Var Table

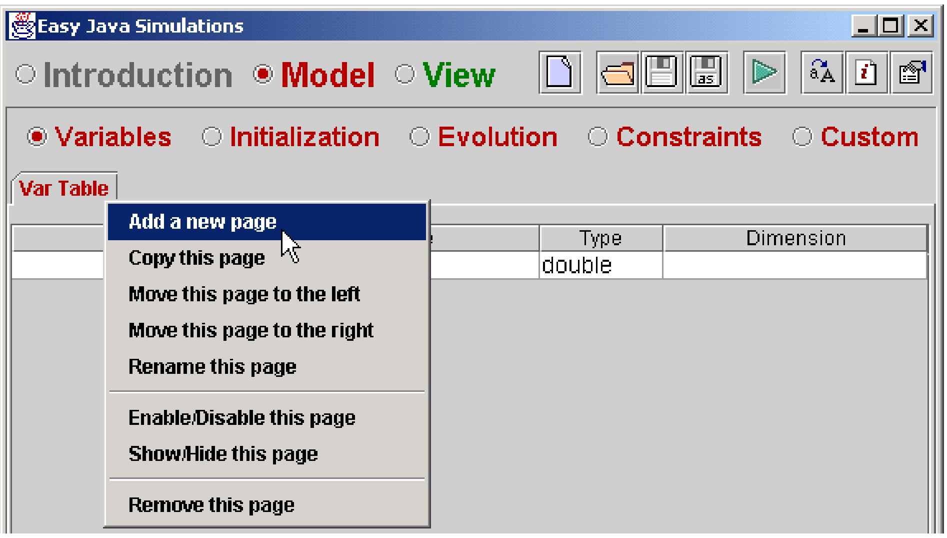
Once you type a name for the new page and click *Ok* (or hit return) a new empty page will be created. Because there can be more than one pages in each of the areas, the pages are organized using a tab system.



Name	Value	Type	Dimension
		double	

This tab system is very convenient when you have a complex model, so that you can organize your variables or code in separate pages, each of them holding variables or pieces of code that have a similar function. Obviously, simple simulations can be described with just one page per part.

You can create a second page, copy, rename, remove, disable and hide this page using a popup menu that appears when you click the right button of the mouse over the top of the page (anywhere in the gray area between the five radio buttons and the white text area of the action's code).



Also, when you already have more than one page, you can use the options in this menu to reorganize the tabs changing the order of the pages.

Enabling and disabling requires a bit of explanation. When you are creating a model it may well be that you want to test several algorithms to solve a given problem and to see which one works best. Of course, you can always try each of them by deleting the previous and writing the new one. But if you want to come back to the one you just deleted, you would have to delete the new one and retype the older. And this once and again, until you decide…

Not with **Ejs**! What you can do instead is to write both algorithms for once in different pages and enable and disable the pages in turn to test either one or the other. Disabled pages are easily recognizable because they append a (D) suffix to their tabs name and can not be edited.

Finally, the option to show and hide a page allows you to keep kind of 'secret' pages of code. This may be useful for pedagogical means when you are creating complete simulations for your students, but you want them to proceed throught them little by little, so that they can grasp the main ideas more easily.

To this end, you could place some not-so-relevant code in hidden pages and configure **Ejs** so that it doesn't show the hidden pages[22]. The simulation would keep full functionality (hidden pages are invisible, but active) and still show your students a simplified part of the underlying model.

Later, when the students are ready for it, you can always tell them to switch **Ejs** to show hidden pages so that they can see how you created the simulation in all its details.

Hidden pages (when made visible) display an (H) sufix to their tabs names.

The remainder of this chapter is devoted to describe how to use each of the panels to complete the steps of section 5.1 for the creation of the model.

5.3 Declaring variables

This corresponds to step 1 above. Variables are easy to define. We only need to give them a unique valid name (see subsection 5.3.3 below), specify the type of the variable (I explain this in a moment) and, in case it is a vector or a matrix, indicate its dimension. I'll refer to variables with dimension as *arrays*.

[22] See section 4.1.1.

5.3.1 Types of variables

Even when in mathematical formulas, variables are usually described using real numbers (occasionally, complex numbers), when we write a computer program we distinguish among several types of variables, depending on the use we make of the variables and on the computer memory needed to store them.

For instance, variables that only take integer values need less memory and can be handled faster in computations, since most computer systems implement optimized routines for integer arithmetic.

We may also want to use variables of type *boolean*, which can only take two values: *true* or *false*, of type *char* to store characters and of type *String* to store text.

In our case, Java implements the following types of basic variables: *boolean*, *byte*, *char*, *short*, *int*, *long*, *float*, *double* and *String*. Where *byte*, *short*, *int* and *long* are used for integers and *float* and *double* for real numbers.

However, except in cases where it is absolutely essential to make the computations always with the smaller possible types (to save space or optimize execution speed to its limits) we can just choose to use the standard types in each category.

This will be our approach and **Ejs** will only make use of variables of the types *boolean*, *int, double* and *String*.

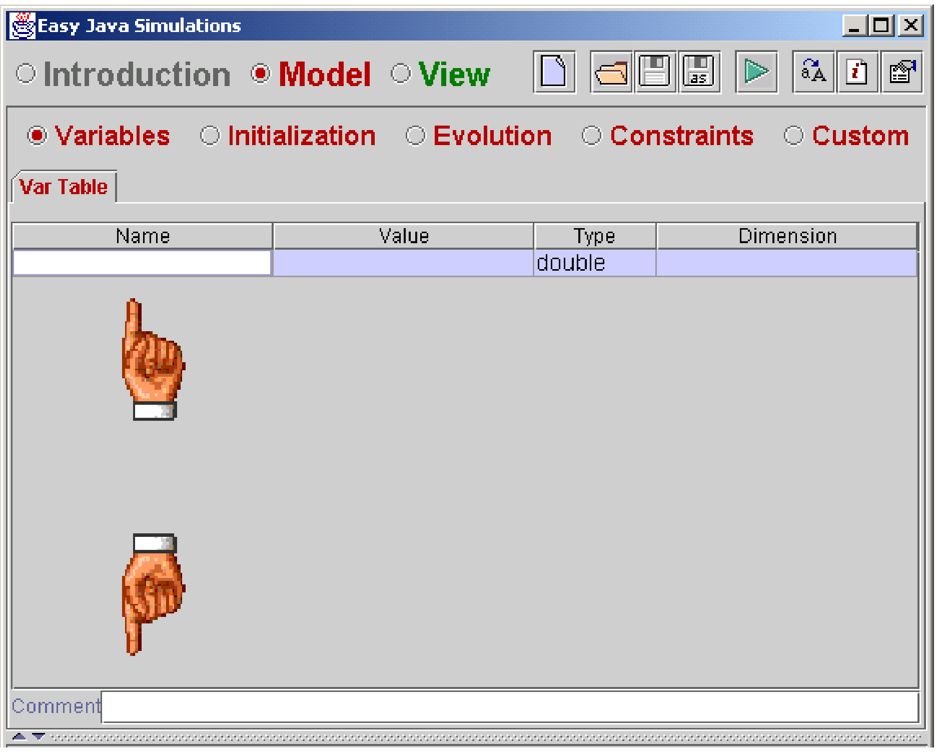
Advanced topic: variables of type Object

Besides these basic types, Java uses a new type called *Object* as the base type for a whole family of advanced variables. Java is an *Object-Oriented* programming language and, although it is out of the scope of this manual to describe what this fully means, this allows Java programmers to create a wonderful and extensive world of new variable constructions, called *classes* that implement all kinds of functionality.

Ejs wants to be an easy tool for a precise task, that of creating model-focused scientific simulations. However, as a door for advanced programmers to use and also to allow us, simple mortals, to create nice visualization tricks (for instance, changing colors dynamically), Ejs introduces in this version the possibility of declaring a new type of variables, *Object* variables. It is however the responsibility of the user to use object variables appropriately. We refer the reader to the examples to appreciate valid, simple uses of this.

5.3.2 Creating variables

Every page of the *Variables* panel holds a copy of the editor of variables, which adopts the form of a table.


Easy Java Simulations
○ Introduction ● Model ○ View
● Variables ○ Initialization ○ Evolution ○ Constraints ○ Custom
Var Table
Comment

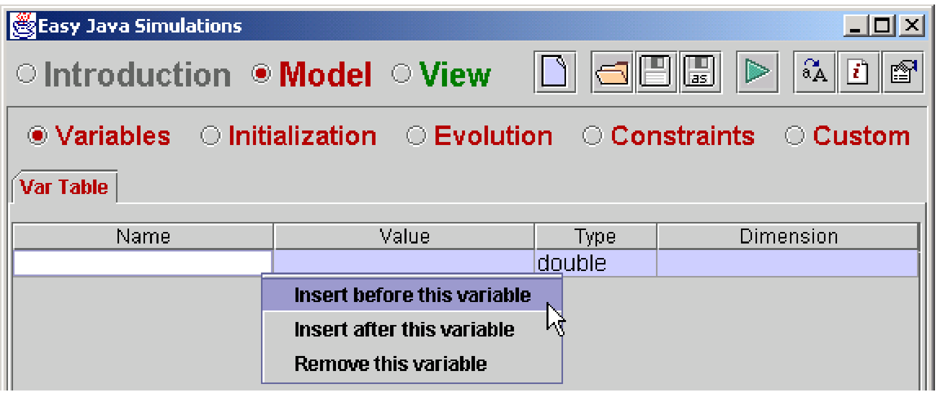
Name	Value	Type	Dimension
		double	

We add a variable to a table by typing the name, providing an initial value, selecting the type and, if the new variable is an array, indicating the dimension of it. We can also type a descriptive comment for each variable in the lower *Comment* field. This comment has no other use but to provide a short explanation of the role of this variable in the model, in case another person reads our model.

Ejs accepts both simple or multidimensional variables. If we leave the field at the column *Dimension* empty, then the variable created is a simple one. A non-empty dimension consists in one or more occurrences of matching square brackets with an integer value in between.

For instance, if we write *[50]* in the *Dimension* field of a variable, this means that we want to declare (what is called in Mathematics) a vector with *50* coordinates. If we write *[10][100]*, this creates a matrix with *10* rows, each of them with *100* elements.

When a variable is created, an empty line is automatically added to the table, thus providing space for more variables. We can also insert a variable between two existing ones, by calling a popup menu, clicking the right button of the mouse. This menu is also used to remove variables.


Easy Java Simulations
○ Introduction ● Model ○ View
● Variables ○ Initialization ○ Evolution ○ Constraints ○ Custom
Var Table
Insert before this variable
Insert after this variable
Remove this variable

Name	Value	Type	Dimension
		double	

The field under the column header *Value* is also optional. Although there is a dedicated panel for initialization of a simulation, sometimes initializing a variable reduces to giving to it a constant value or the result of a simple expression. In this case, it is not necessary to create an initialization page (as we will learn to do in the next section) for so little, you can just type the variable value in the *Value* column. If you use this feature for an array, all its elements will get the same initial value.

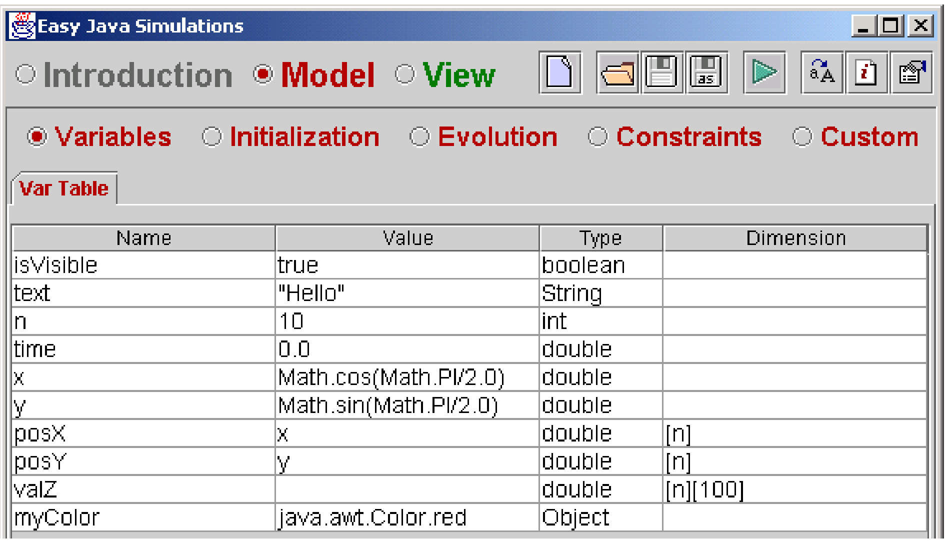
Leaving this field empty will always imply a default initialization to *0* for numeric variables, to *false* for booleans, to an empty string “” for Strings and to *null* for Objects.

For example, the definition of the following variables:

- *isVisible* : *boolean*, simple, initialized to *true*.
- *text* : *String*, simple, initially set to “*Hello*” [23].
- *n* : *int*, simple, initialized to *10*.
- *time*, *x* and *y* : *double*, simple, initialized to *0.0*, *cos(π/2)* and *sin(π/2)*, respectively.
- *posX* and *posY* : array of *n* *doubles*, all of its elements initialized to *x* and *y*, respectively.
- *valZ*: bi-dimensional array of *n* × *100 doubles*.
- *myColor* : *Object*, simple, initialized to red color.

appears in the editor as follows:

[23] Notice the inverted commas, constant strings require them.


Easy Java Simulations
○ Introduction ● Model ○ View
● Variables ○ Initialization ○ Evolution ○ Constraints ○ Custom
Var Table

Name	Value	Type	Dimension
isVisible	true	boolean	
text	"Hello"	String	
n	10	int	
time	0.0	double	
x	Math.cos(Math.PI/2.0)	double	
y	Math.sin(Math.PI/2.0)	double	
posX	x	double	[n]
posY	y	double	[n]
valZ		double	[n][100]
myColor	java.awt.Color.red	Object	

As this example shows, it is possible to use a variable in the definition of another, as long as the first one is defined and initialized before it is used. Please notice also the correct way of using the mathematical cosine and sine functions (and the constant π) in Java. Notice too how I initialized the Object myColor to the color red.

This uses one of the (literally) hundreds Java classes that exist, and which I have learnt to use by reading some Java books. The bad news is that this manual is not the place to learn Java. The good news is that, in my opinion, this class, called *java.awt.Color* is perhaps the only one that really adds a feature that we do need for our purposes, that of dynamically changing the color of our view elements.

You can learn a bit more about some interesting Java classes, including *java.awt.Color*, in the appendices.

When, later on, we want to use a simple variable in a Java expression, we only need to write its name. When using an array, however, we must also indicate which element of the array we are referring to. We do this by writing the name of the array followed by the index between square brackets, where the first element has zero index and the last element has an index equal to the size of the array minus one.

> It is a common (and dangerous!) mistake to refer to the first element of the array *posX*, in the example above, as *posX[1]* and to the last one as *posX[n]* or *posX[10]*. The correct way would be *posX[0]* and *posX[n-1]* or *pos[9]*, respectively.

Correct examples of use of variables in Java code include the following:

```
isVisible= false; t = 3.5; posX[0] = 1.0; posY[9] = 4.2; valZ[9][99] = 5.0;
```

while the next are incorrect:

```
x[0] = 0.1; posX[10] = 1.0; posY[0][0] = 1.0; valZ[99][9] = 5.0;
```

5.3.3 Naming conventions

Along the process of creating a simulation we'll need to provide names for several elements of it: for pages in different parts of the simulation, for variables, custom methods or actions of the model, and for elements of the view.

The names for pages have no special use but to help you identify the different parts of a simulation, so you can use any name for any page, with no restrictions. You could even use the same name for different pages.

But the names for the other elements must follow some basic rules, in order to avoid conflicts among these names as well as to make the simulation easier to understand for other people. We will follow the following naming conventions:

1. The name of every variable, method or view element must be unique in the whole simulation.
2. Names are formed by a conjunction of alphanumerical characters (from a to z, from A to Z, and from 0 to 9) with no limitation in length.
3. The first character must always be an alphabetic character. For variables and methods the first letter will be chosen in lowercase (that is, from a to z). For view elements the first character can be written uppercase.
4. It is recommended to use descriptive names. To this end, several words can be put together (with no blank spaces in between) to form a name. In this case, it is better to start each new word with an uppercase letter.
5. Names starting with an underscore character '_' are forbidden, since Ejs uses them for its own variables and methods.

Finally, the following words are reserved **Ejs** or Java keywords; hence, they can not be used as names.

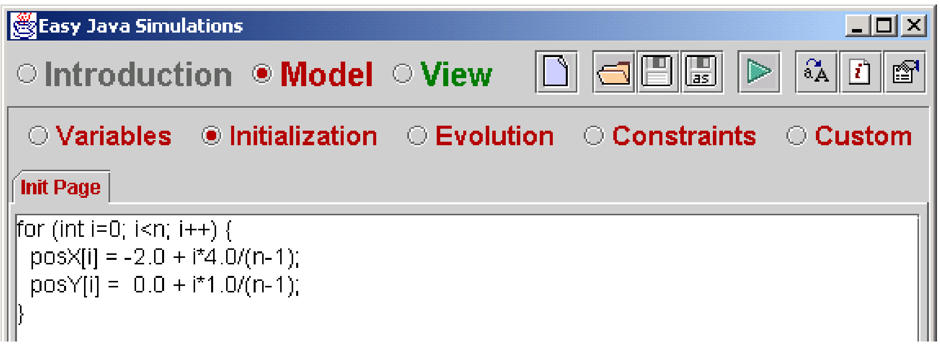
boolean, break, byte, case, catch, char, continue, default, do, double, else, float, for, getSimulation, getView, if, initialize, instanceof, int, long, Object, reset, return, short, step, String, switch, synchronized, throws, try, update, while.

5.4 Initializating the model

As we have seen, we can initialize the variables to constant values or to simple expressions when we define them. However, more complex initializations are often needed. For instance, if we want to give different values to the elements of an array, or assign to a variable a value that has to be computed using an elaborated mathematical algorithm.

To do this, we use the second of the panels of the model, the one labeled *Initialization*. Here, we will create the pages of Java code needed to initialize the simulation[24]. The code in these pages will be executed once, and only once, at the beginning of the simulation, in the same order as their tabs show.

As an example, a page that initialized the arrays *posX* and *posY* that we defined in subsection 5.3.2, with equally spaced values from the intervals *(-2,2)* and *(0,1)*, respectively, would look like the following.

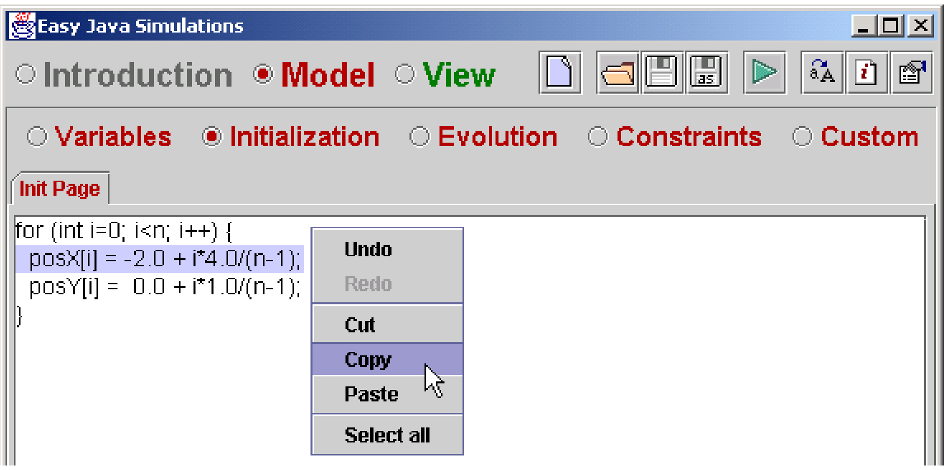

Easy Java Simulations
○ Introduction ● Model ○ View
○ Variables ● Initialization ○ Evolution ○ Constraints ○ Custom
Init Page

```
for (int i=0; i<n; i++) {
  posX[i] = -2.0 + i*4.0/(n-1);
  posY[i] =  0.0 + i*1.0/(n-1);
}
```

The initialization of a model is, in principle, that simple. You can add as many pages as you want with the appropriate Java code on it. You need to remember that, if there are more than one initialization page, these are executed exactly in the same order they appear in this panel, from left to right. You can, however, change this order by using the corresponding options in the popup menu for this panel.

The text area where you write the code has also its own popup menu. This contains the basic editing capabilities to *undo*, *redo*, *cut*, *copy*, *paste* and *select all* text.

[24] You can find a reminder of the basic Java you need in order to write computer algorithms in the appendices.



○ Introduction ● Model ○ View

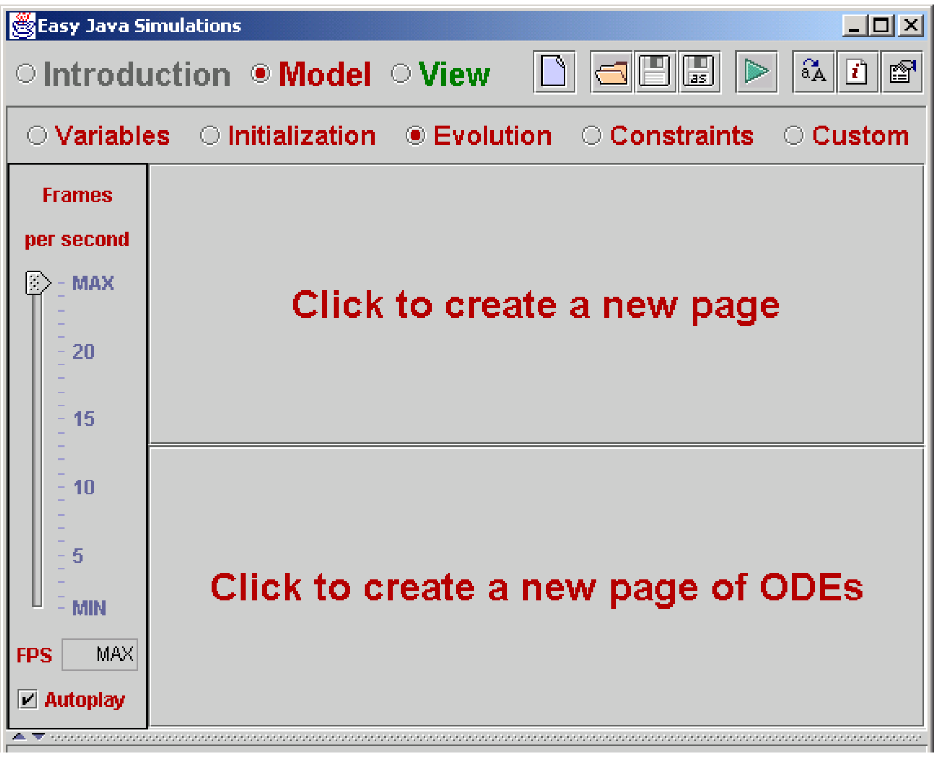
○ Variables ● Initialization ○ Evolution ○ Constraints ○ Custom

```
for (int i=0; i<n; i++) {
  posX[i] = -2.0 + i*4.0/(n-1);
  posY[i] =  0.0 + i*1.0/(n-1);
}
```

You can also select part of the text by (left) clicking and dragging the mouse over the desired piece of text. Although drag and drop is not supported (at least not yet) you can use the standard acceleration shortcuts for cut (Control-X), copy (Control-C) and paste (Control-V). The figure above shows the process of selecting a piece of code and copying using the popup menu.

5.5 Evolution equations

The interface for this panel is a bit more elaborated.



○ Introduction ● Model ○ View

○ Variables ○ Initialization ● Evolution ○ Constraints ○ Custom

FPS: MAX

☑ Autoplay

Easy Java Simulations
Frames
per second
MAX
20
15
10
5
MIN
Click to create a new page
Click to create a new page of ODEs

To the left of the card, we see a slider that modifies a value called *Frames per second*, followed by a non-editable text field labeled *FPS* and a checkbox which reads *Autoplay*. We will explain the utility of these controls in a minute.

Unlike previous panels, the main central part of the panel is now divided in two subpanels. The upper one invites us to create a new page, the lower one invites us to create a new ordinary differential equation (or ODE). This corresponds to the fact that the evolution of a model can be described in **Ejs** in two different, complementary ways.

In principle, the implementation of evolution equations consists simply in transcribing them into sentences of Java code. Sometimes, however, evolution equations derive from the numerical resolution of systems of ordinary differential equations, a task that implies writing complex code.

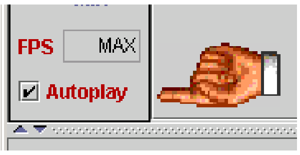
Although you might prefer to write the code for solving ODE by yourself, **Ejs** includes the possibility of introducing these equations in a dedicated editor and then, it automatically generates the associated code to solve the equations using the most popular solving algorithms.

Hence, we would click on the upper message to start with a blank area for Java code, and on the lower one to start with an ODE editor. Later, we can always add both, pages of code or of ODE, using the usual popup menu.

5.5.1 Setting the execution environment

But let us first describe what the left controls mean. We start from the bottom.

By default, when a simulation runs, the evolution is started automatically. This is what the tick that you see on the *Autoplay* checkbox means. But, sometimes, you may want to change this default behavior.



For instance, suppose that you want the user to manipulate the simulation in order to complete certain tasks before the evolution starts. Then, you need to uncheck the *Autoplay* checkbox and the simulation will show up but will not start the evolution. Of course, in this case, you will need to provide your user with a way of starting the evolution, once he or she completes the required task.

> The standard way for this is to include a button in the view with its *Action* property associated to the predefined method _ *play()* (see chapter 6.)

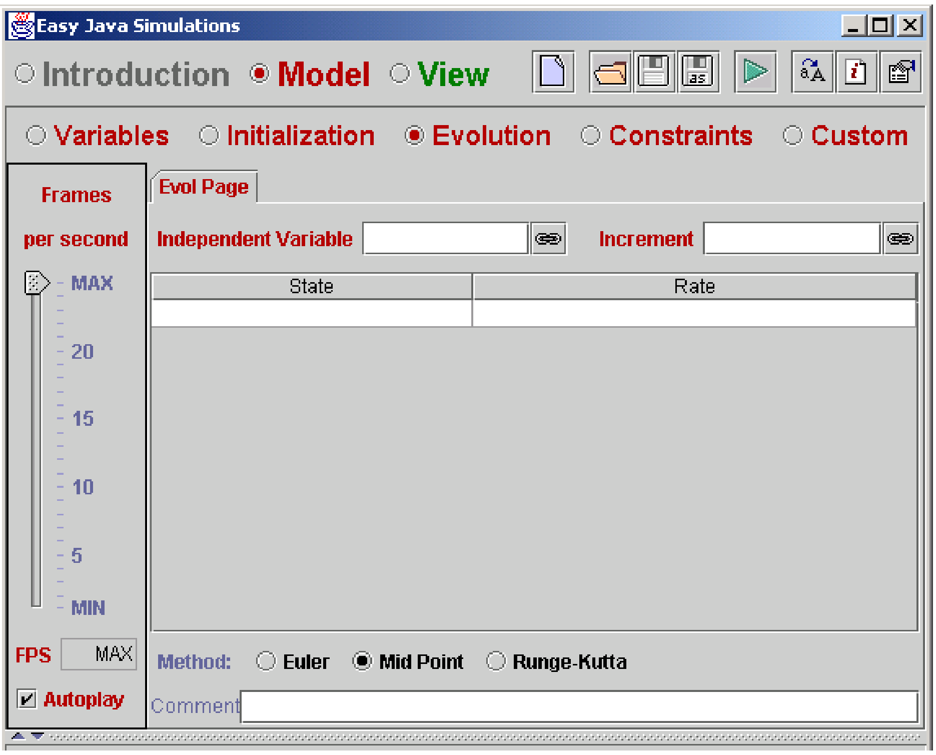
Moving upwards from the *Autoplay* checkbox, we see the controls for the speed of the simulation. Usually, you might expect the computer to run the simulation as fast as it can, repeating evolution steps over and over with no interruption nor delay. However, because modern computers are becoming faster and faster, it is sometimes necessary to force them to wait a little between every two successive evolution steps. Otherwise the evolution is so fast that we cannot appreciate what is happening.

This is the purpose of the *Frames per second (FPS)* control and field. The *FPS* value represents the approximate number of evolution steps that the simulation should complete in one second. The minimum is *1* and the biggest numerical value is *24*. There is still a special non-numerical value, which reads *MAX*, which means 'as fast as you can'. This is the default value.

5.5.2 Writing equations

Let us now go back to our equations. If, as said before your evolution equations can be described using regular Java code (and one can build really sophisticated models with it!) you will just add pages of code in the same way, and with the same utilities as you did for your initialization pages. If you have ODE that you want to solve by yourself, you can also proceed as you learnt in the previous section.

The only think I need to describe is the editor for ODE, in case you want to use it to help you solve (numerically) your ODE. This editor is displayed in the next picture.


Easy Java Simulations
○ Introduction ● Model ○ View
○ Variables ○ Initialization ● Evolution ○ Constraints ○ Custom
Frames per second
MAX
20
15
10
5
MIN
Evol Page
Independent Variable
Increment
State
Rate
Method: ○ Euler ● Mid Point ○ Runge-Kutta
FPS MAX
☑ Autoplay
Comment

In order to illustrate the use of the editor with a practical example, I will use the same sample variables that we defined in section 5.3 . We will define two systems of ODE using these variables. The first one will be the second order differential equation $x''(time)+x(time)=0$, which, using y as an auxiliary variable turns into

$$\begin{cases} x'(time) = y(time) \\ y'(time) = -x(time) \end{cases}$$

Of course we could easily find the analytical solution to be of the form $x(time) = A\cos(time) + B\sin(time)$. But we will ignore this possibility now (or loose our example!).

The second will be a similar system of ODE but for each of the elements of the arrays *posX* and *posY*. That is, in vector notation,

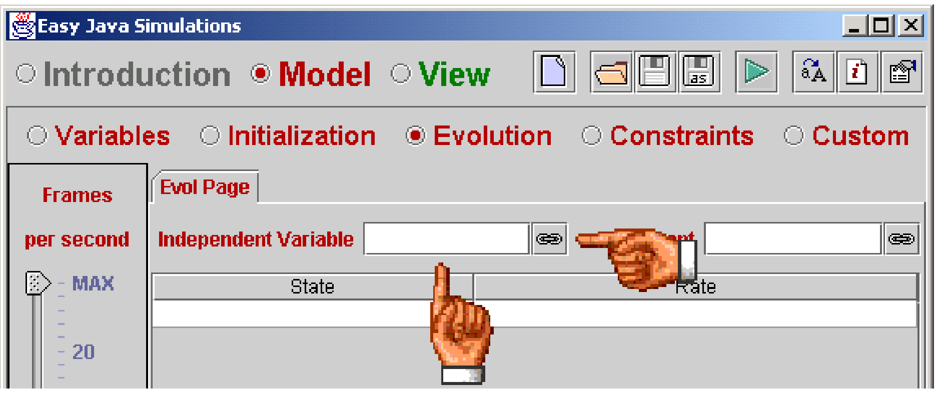
$$\begin{cases} \mathbf{posX}'(time) = \mathbf{posY}(time) \\ \mathbf{posY}'(time) = -\mathbf{posX}(time) \end{cases}$$

This possibility of defining ODE for arrays is a very powerful one, since it allows you to generate simulations with a large number of individual variables with a relatively small effort.

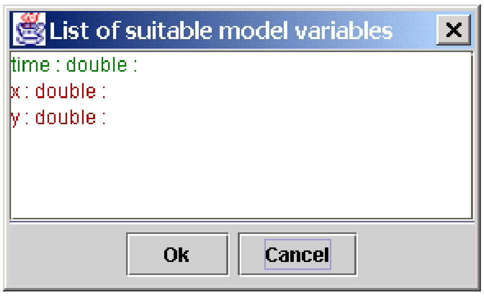
5.5.3 Declaring an ODE

The first thing we have to do is to choose the independent variable for the ODE system. In our set of sample variables, this will be *time*.

We can do this in two different forms. The first one is to simply type *time* in the field labeled *Independent Variable*. The second is to click in the ⚭ icon to the right of this field.

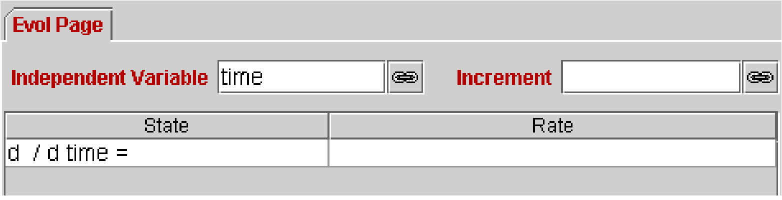

Easy Java Simulations
○ Introduction ● Model ○ View
○ Variables ○ Initialization ● Evolution ○ Constraints ○ Custom
Frames
per second
MAX
20
Evol Page
Independent Variable
State
Rate

If we click on it, a dialog will show up, listing all the variables of the model that can be used in this field.



time : double :
x : double :
y : double :

We can then select the variable *time* and click *Ok*. The *Independent Variable* field will reflect the change.



Evol Page

Independent Variable: time

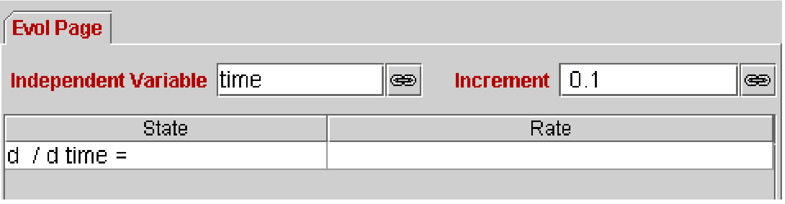
Increment:

State	Rate
d / d time =	

You will also notice that the State column of the table below has changed and reads not ‘*d / d time*’. It is ready to display your differential equation.

The second thing we need to do is to specify the increment for the independent variable in each evolution step. This corresponds to the step that we will use in the numerical method and can be either a variable from our list or a constant value. In the first case, we can use any of the methods described above to indicate our choice. If we use a constant value, we just simply type it in the field.

For our example, we will use the constant value *0.1*. So we just type it.



Evol Page

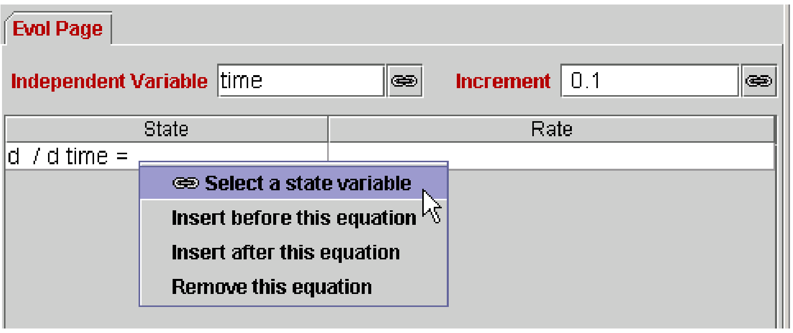
Independent Variable: time

Increment: 0.1

State	Rate
d / d time =	

Finally, we must write the differential equations of the system. This is done selecting in a row of the table, for each of the equations in the system, the state variable that we want to derivate with respect to our independent variable (in the column with the header *State*) and, to its right (in the column with the header *Rate*), the value of the corresponding derivative.

Again, to indicate each state we can simply type in the variable name (in the space between the first ‘*d* ‘ and the ‘*/ d time*’) or (recommended) select it from a list of suitable model variables. This list is shown in a dialog that appears when we select the option *Select a state variable* in the popup menu for a given row.

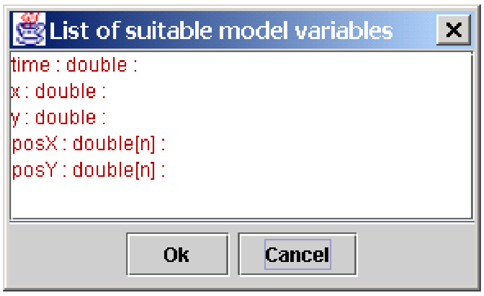


Evol Page

Independent Variable: time

Increment: 0.1

State	Rate
d / d time =	



time : double :
x : double :
y : double :
posX : double[n] :
posY : double[n] :

This time, however, there is a difference, since **Ejs** allows that the state is either a simple or a uni-dimensional double variable.

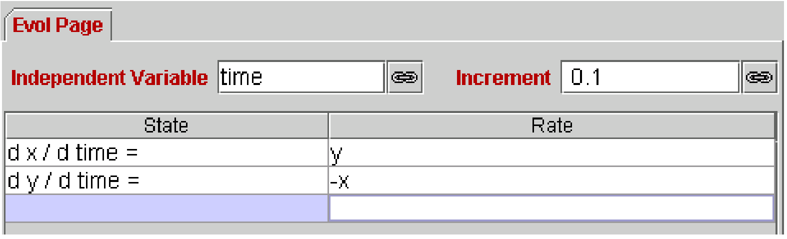
Hence, all variables with type double, both simple and uni-dimensional, will be shown in this dialog.

The expression for the derivative (or rate) of this state variable must be typed in explicitly. Just click on the corresponding field and type the expression. Notice that this expression can contain any of the variables of the model, not just those shown on the list above.

In our case, we select *x* and *y* as state variables and type *y* and *–x* as rates, respectively. This corresponds to the second order differential equation

$$\begin{cases} x'(time) = y(time) \\ y'(time) = -x(time) \end{cases}$$

The result looks like this.



Independent Variable: time
Increment: 0.1

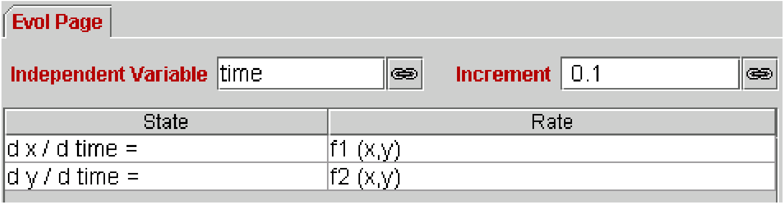
State	Rate
d x / d time =	y
d y / d time =	-x

IMPORTANT NOTICE

If the description of the equation is a bit complicated, it is very common (and useful) to write in the right-hand side of the table the call to a Java method that we define previously in the *Custom* panel of the model (as described in section 5.7). This keeps the ODE easier to read.

When we do so, however, it is very important to make this expression ‘self-contained’. By this I mean that, if the rate expression depends on any of the state variables of this systems of ODE (as it usually does), then these states must be included as parameters of the custom function. The technical reason for this relies in the way some numerical algorithms use intermediate states to improve the accuracy of the computation.

If we wanted to do so for our equations, the safest way to do it would be the following:



Independent Variable: time   Increment: 0.1

State	Rate
d x / d time =	f1 (x,y)
d y / d time =	f2 (x,y)

declaring in the *Custom* part (you may need to wait until you read section 5.7 to get a full understanding of what is described here) the methods

```
private double f1 (double a, double b) { return b; }
private double f2 (double a, double b) { return -a; }
```

Notes:

1. Since the name of the parameters are generic, I like to use different names than that of the real variables, just to avoid confusion.
2. These methods can be declared *private* since they are going to be used only by the model itself.

We now add to the system the equations for the second ODE

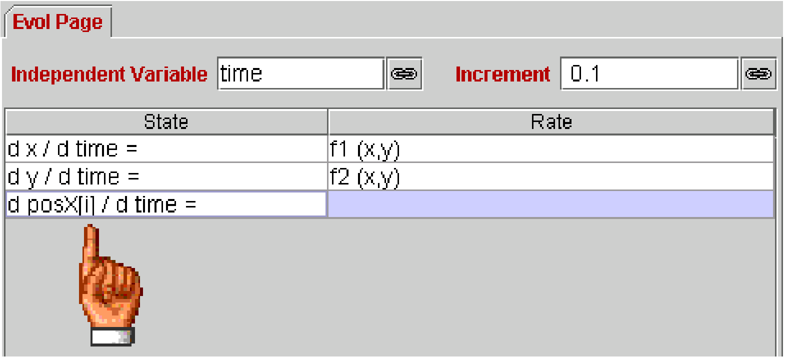
$$\begin{cases} \mathbf{posX}'(time) = \mathbf{posY}(time) \\ \mathbf{posY}'(time) = -\mathbf{posX}(time) \end{cases}$$

Since the independent variable is the same, we can (in fact, should[25]) use the same page we already created. The procedure is similar to the one we used for the case of simple variables.

The only difference in describing the second ODE is that it applies to all the elements of the array. Hence, we have to indicate the rate for each of these elements, and this rate usually depends on the index of the element.

[25] If we used a second page with the same independent variable and increment, the time would be incremented by 0.1 twice in each evolution step, once per page. And this could affect the accuracy of the numerical algorithms for solving the ODE.

For this reason, if we select a unidimensional state variable, **Ejs** will add an index (between square brackets) to the name of the variable.



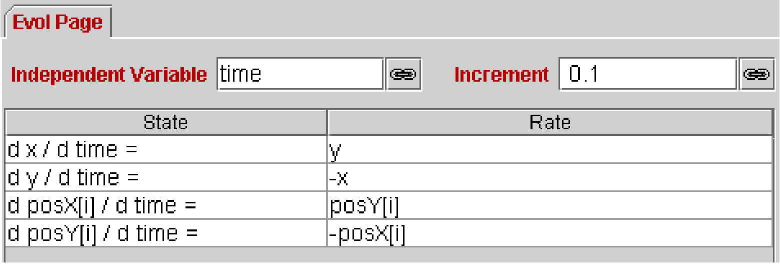
Independent Variable time

Increment 0.1

State	Rate
d x / d time =	f1 (x,y)
d y / d time =	f2 (x,y)
d posX[i] / d time =	

By default this index is *i* but you can change it to any other variable, even if you have not defined it in the tables of variables. Then, when writing the rates, you should use the same index in the expression, if the rate depends on it.

In our example, the final result should look like this.

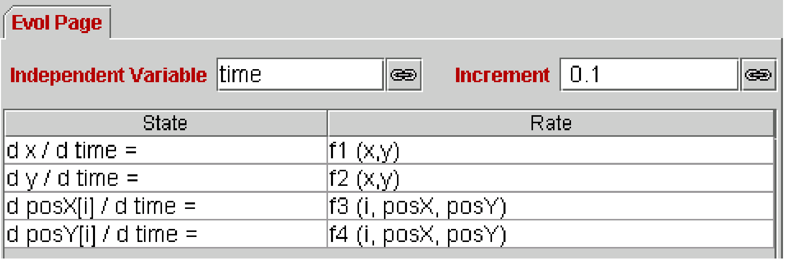


Independent Variable time

Increment 0.1

State	Rate
d x / d time =	y
d y / d time =	-x
d posX[i] / d time =	posY[i]
d posY[i] / d time =	-posX[i]

Continuing the notice above, if we want to use custom methods the right way would be, in this case, the following:



Independent Variable time

Increment 0.1

State	Rate
d x / d time =	f1 (x,y)
d y / d time =	f2 (x,y)
d posX[i] / d time =	f3 (i, posX, posY)
d posY[i] / d time =	f4 (i, posX, posY)

adding to the *Custom* part the methods

```
private double f3 (int i, double[] A, double[] B) { return B[i]; }
private double f4 (int i, double[] A, double[] B) { return -A[i]; }
```

5.6 Writing constraint equations

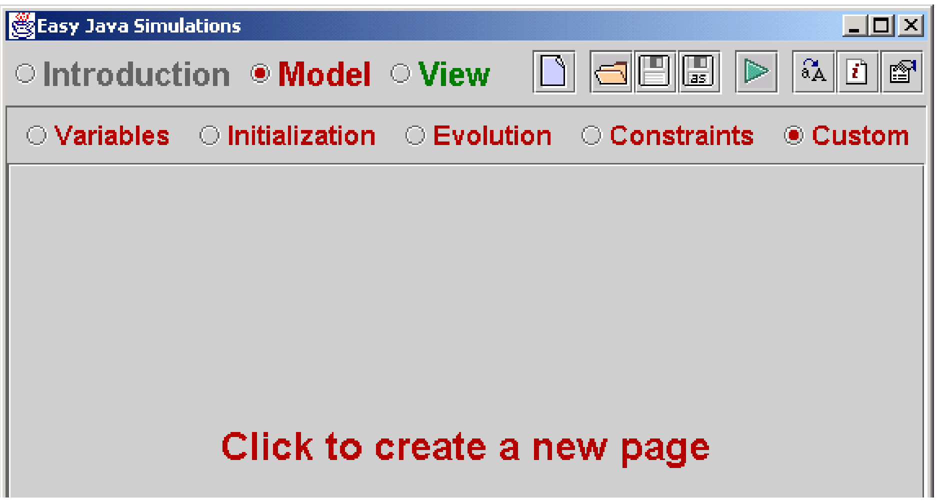
Creating pages for constraint equations presents an interface identical to the *Initialization* panel, though their effect are different according to what is described in subsection 5.1.3.

We only need to write the code that transcribes the constraint equations into sentences of Java language.

5.7 Custom methods

The last radio button of the *Model* part of the interface of **Ejs** is labeled *Custom*. Its purpose is to let you define your own methods (functions or subroutines) so that you can use them anywhere else in the simulation.

The interface and purpose is similar to that if the initialization and constraint pages,


Easy Java Simulations
○ Introduction ● Model ○ View
○ Variables ○ Initialization ○ Evolution ○ Constraints ● Custom
Click to create a new page

but have two important differences.

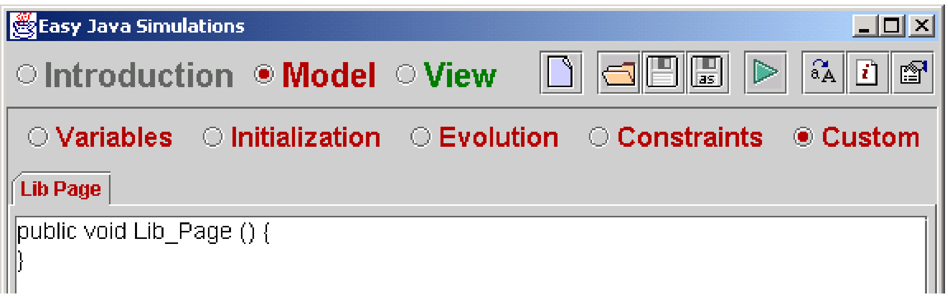
The first one is that methods created in *Custom* pages have to be explicitly called by you in any other part. Unlike *Initialization* pages, which are called at start up, *Evolution* pages, which are invoked at every evolution step and *Constraint* pages, called whenever the model state changes, *Custom* pages are not called by the simulation environment. You have to explicitly include a call to the methods you define here in any of the other part of the simulation.

The second difference is that you have a greater degree of freedom to create methods in these pages. More precisely, while other pages create internal Java methods that can not be called directly, and that can not be called with parameters, in this ocassion you can create methods that accept parameters and return a value (thus behaving like functions).

This offers you more freedom, yes, but also requests from you that you know how to declare valid Java methods. I'll describe briefly how to do this in the next subsection.

5.7.1 Creating your own methods

When you create a new page, **Ejs** gives you a hint by providing an initial basic skeleton.


Easy Java Simulations
○ Introduction ● Model ○ View
○ Variables ○ Initialization ○ Evolution ○ Constraints ● Custom
Lib Page

```
public void Lib_Page () {
}
```

Creating a Java method requires that you give it a name, a return type and that you declare its accessibility. Optionally, we can also declare the method to accept input parameters.

The term 'accessibility' refers to whom can use the method. If you declare it *public*, any other part of the simulation, or even utilities outside the simulation (like JavaScript, see chapter 4), can use it. In general, it doesn't hurt to make all your methods public.

If you declare it *private*, however, then nobody but the model can use it. The only reason I see to declare methods private is to keep the list of methods offered to view elements (for linking with their action properties, see next chapter) shorter.

The return type applies to methods that return a value as result of their internal operations. If a method doesn't need to return a value, it can be declared *void*. If it does, then the last sentence of the method's logic must be a *return* statement.

Finally, parameters consist in a comma-separated list of pairs *type name* that declare variables that will be used in the method internal body (the code we write between the curly braces). If a method accepts parameters, then it must be called with values that match the type of the declared list of parameters (see the example below). If a method needs no parameters, it can be declared with an empty list of parameters.

You can change anything in the basic declaration proposed by **Ejs**: the accessibility, the return type, the parameter list (which is initially empty) and even the name of the method, since the name you give to the tab is taken initially to provide a name for your method but afterwards is only used to decorate the tab.

You can also declare more than one method in each page.

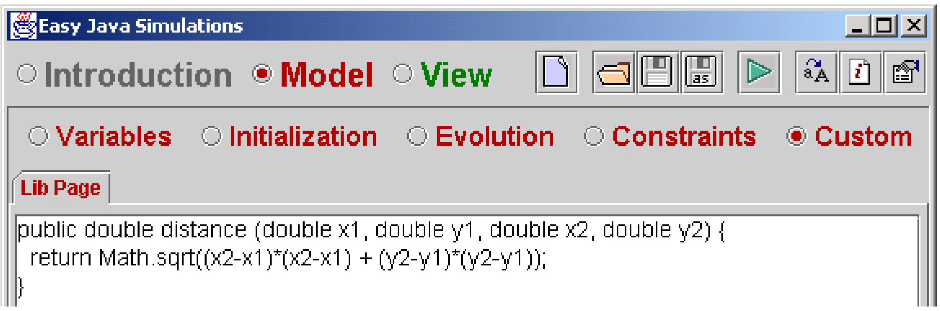
The typical use of *Custom* pages is to define methods that include algorithms that are rather elaborated and that, if written in the place where they are used, could make reading (and understanding) the simulation's logic more difficult.

A second typical use is to write algorithms that are going to be used in more than one place of the simulation and with different values or parameters. This improves reusability of your own code.

We will illustrate the use of this panel by declaring a method that computes the distance between two points and then use this method in the constraint pages.

The distance between two points in the plane, *(x1,y1)* and *(x2,y2)*, is given by the formula $d((x1,y1),(x2,y2)) = \sqrt{(x1-x2)^2 + (y1-y2)^2}$.

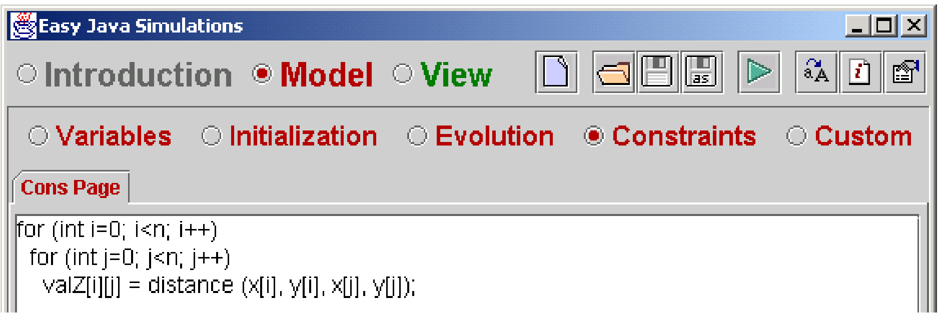
To turn this formula into a valid Java method we change the code of our initial method *Lib_Page()* to read,



```
public double distance (double x1, double y1, double x2, double y2) {
  return Math.sqrt((x2-x1)*(x2-x1) + (y2-y1)*(y2-y1));
}
```

Notice that we have changed the return type to double, the name to something meaningful according to the purpose of the method, and the parameter list to accept the coordinates of our two points as input.

Now, we can use the newly created method in a constraint page



```
for (int i=0; i<n; i++)
  for (int j=0; j<n; j++)
    valZ[i][j] = distance (x[i], y[i], x[j], y[j]);
```

to keep *valZ* updated with the value of the distances of each point *(posX[i],posY[i])* to the others[26].

[26] Yes, I know this code is not very elegant, nor efficient: *valZ* has a lot of unused space and the computation is redundant since the distance from one point to another and back are the same. But the intention of the example is just to illustrate the use of **Ejs**' interface.

5.7.2 Defining control actions

Besides using this *Custom* part to create utility methods that help you simplify the coding of other parts in the model, we can also use this panel to prepare control actions that can be used to provide your simulation with a certain interactivity.

Later, when we create a view for our simulation, we can include in it elements that will help the user trigger these actions (by clicking on a button, or dragging the mouse, for instance).

Creating control actions consists in defining a series of methods that the user may want to call to modify the behaviour of the simulation.

Control actions are created exactly in the same way as you create any other custom method. There are however some restrictions. For technical reasons (that would be very long to describe here) control actions,

1. must be *public*,
2. do not need to return any value (although they can do so, if you want), and
3. must be declared with either an empty parameter list or with, at maximum, one single parameter. In this later case, this parameter can only be a *boolean*, an *int*, a *double*, or a *String* (arrays, for instance, are not allowed).

Although it may seem a bit restrictive, I do believe that this can serve most, if not all, of your needs.

Typically, the code of an action contains sentences that change the value of the variables of the model. Hence, their purpose is to set a given state of the model, which is of special interest for the user.

Also, in some occasions, the user may also want to change the view: for instance, to show or hide a given component of the visualization. Changing the view is always done in a standard way; since you can link some properties of the elements of the view to model variables (as we will see in the next chapter), you only need to change these variables to make the view behave the way you want[27].

Finally, you may also need actions to control the execution of the simulation itself; for instance, to start and stop the evolution, to modify the execution

[27] See, however, the predefined action *_clearView()* in next subsection.

speed, to run the simulation step by step or to reset the simulation to its initial state. You don't need to care about creating these ones; they are provided by **Ejs** itself.

These predefined actions can be called either directly or as part of your own, more elaborated actions (for instance, you may want to stop the simulation, do some computation, clear the view and then restart the simulation). These actions are described in the next subsection.

5.7.3 Predefined actions

Ejs has some predefined Java methods that are very useful to control the execution of the simulation. It also has some methods that allow saving the state of the model to memory or to the local disk and reading it later either from memory, from the local disk or from a web server. This can be very useful if you want to prepare experiments that require an elaborated initial state of the model, in order to present it later to students.

Since Java applets cannot write to the local disk, if you want to save the state of a simulation you have to run it either from **Ejs** or as an independent application (see chapter 4).

To use any of the following methods, you only need to call them from your actions or link them to the actions property of an element of the view.

> For backwards compatibility, these predefined methods have a second way of calling them, one that makes its class (say, its family) explicit. For instance, the first method could be called either as *_play()* or as *_simulation.play()*.
>
> I recommend using always the simplified method. However, I make this remark here in case you ever come across any of these while reading a simulation created with a previous release of **Ejs**.

The following table lists these methods and their corresponding effect.

Methods to control the simulation execution
void **_play** ( ) (or **_simulation.play** ( )) Starts the evolution
void **_pause** ( ) (or **_simulation.pause** ( )) Stops the evolution
void **_step** ( ) (or **_simulation.step** ( )) Executes one step of the evolution
void **_setFPS** (int fps) (or **_simulation.setFPS** (int fps)) Sets the (approximate) number of evolution steps per seconds to the prescribed value

void **_setDelay** (int delay) (or **_simulation.setDelay** (int delay))

Sets the delay for the evolution to the prescribed value. This can be used to control, more precisely than with *setFPS* or **Ejs**' *Frames per seconds* slider the speed at which the simulation should run. This method instructs **Ejs** to wait, after completing an evolution step, the number of milliseconds indicated by the integer value *delay*, before starting the new one.

void **_reset** ( ) (or **_simulation.reset** ( ))

Resets all model variables to their initial values and executes all initialization pages

void **_initialize** ( ) (or **_simulation.initialize** ( ))

Executes all initialization pages, but respects any change to the initial values of the variables that the user has done on the view. A call to *initialize* represents a initialization softer than *reset*, so to say.

boolean **_isPlaying** ( )

Returns true if the simulation is running.

boolean **_isPaused** ( )

Returns true if the simulation is paused.

Methods to save and retrieve the state of the model

boolean **_ saveState** (String filename) (or **_simulation.saveState** (String filename))

Saves the value of all variables of the model in the specified file. If it succeeds, it return *true*, if there is any problem, it returns *false*.

If the filename starts with the prefix *ejs:* then data is saved to a temporary memory location. You can read from this location during the same session of the simulation, as if it were a file, but the data will be lost when the simulation exits. This possibility is useful to temporary store interesting states to which you may want to come back later on (during the same session).

boolean **_ readState** (String filename) (or **_simulation.readState** (String filename))

Reads the value of all variables in the model from a file. If the string *filename* represents a URL location on the Internet, it will try to connect to the corresponding web server and read from it. If it succeeds, it returns *true* and automatically calls constraints equations and updates the view.

Methods to change or read the value of variables using JavaScript

boolean **_ setVariables** (String command, String delim, String arrayDelim)
(or **_simulation.setVariables** (String command, String delim, String arrayDelim))

This method is used to set the value of any variable. It is of special interest when using JavaScript to control the application from an html file (see chapter 4). *command* is a string containing a series of instructions of the form *var=value* separated one from each other by the string delimiter *delim*, where *var* must be the name of a variable of the model and *value* an acceptable value for it. If the variable is a one-dimensional array, then the different values of the elements of the array must be separated by the string delimiter given by *arrayDelim*.

For example:

```
_simulation.setVariables(“x=1.0;y=0.0”,”;”,”,”);
_simulation.setVariables(“matrix=1.0,2.0,3.0;y=0.5”,”;”,”,”);
```

are valid uses of *setVariables*.

boolean **_ setVariables** (String command)
(or **_ simulation.setVariables** (String command))

This is a simplified versionof the previous one. It is equivalent to

```
_simulation.setVariables(command,”;”,”,”);
```

String _ **getVariable** (String varName)
(or **_simulation.getVariable** (String varName))

This is the counterpart of *setVariables*. Given the name of a variable, it returns a string with its value.

Methods to control the view

void **_clearView** ( ) (or **_resetView** ( ))

Clears the view and sets it to its default initial state.

Methods to print in the view (if it contains a TextArea)

void **_print** (String text)

Prints the indicated text in the text area. If the view does not include a text area element, it prints the message to the standard output (usually the console).

void **_println**(String text)

Prints the indicated text in the text area, followed by a line feed and carriage return. If the view does not include a text area element, it prints the message to the standard output (usually the console).

void **_println**()

Prints a line feed and carriage return. If the view does not include a text area element, it prints to the standard output (usually the console).

void **_clearMessages**()

Clears the text area. If the view does not include a text area element, this method has no effect.

6. Creating a view with Ejs

6.1 Graphical interfaces

Without any doubt, creating the graphical user interface is the part of the simulation that demands, if you want to program it directly, more knowledge of advanced programming techniques which, to make the situation even worse, are different from one language to another. It also usually implies long searches in reference manuals looking for the right libraries and graphical routines.

However, the advance of computer graphics makes a simulation without an advanced graphic visualization impossible to conceive. And, if it has to be used for teaching purposes, without a high level of interactivity.

For these reasons, **Ejs** uses a simple, but powerful as we will see, graphic library of Java and has tried to simplify its use as much as possible. This library is based on Java's *Swing* toolkit and on the *Open Source Physics* tools created by Wolfgang Christian at Davidson College[28], but also contains my own contributions. Both set of tools are supported by Java 2 and its plug-in, so they can be visualized by most modern browsers. I will make a special presentation (that may be considered irreverent by purists) of this library that fits our needs: simple and effective… Here we go.

We create a graphic interface for our simulation by building a tree-like structure of selected interface elements.

An *element* is like a piece, or building block, of the interface that occupies a (usually rectangular) area of the computer screen and that performs a particular task on which it specializes. There are elements of several classes: panels, labels, buttons,… and an important part of learning the game consists in knowing which classes of elements exist and how to use and personalize them.

[28] See http://www.opensourcephysics.org

The graphical appearance of each element is mainly determined by its class. However, every element provides some characteristics of itself, called *properties*, which the user can change to fit her needs. Also, some elements have properties of a special type, which I will call generically *actions*, to specify what to do when the user interacts with the element (making a gesture with the mouse or typing the keyboard).

The names of element classes are rather descriptive of their natural use. However, the appendices include a description of all the elements that can be used in **Ejs**, grouped by functionality.

To continue with our general description of elements, we only mention here one main classification among elements: *Containers*, *Basic* elements and *Drawables*.

6.1.1 Containers (and Layouts)

A *container* is a graphic element that can host other elements. If it does, we call the container element the *parent* and the contained element a *child*. Since a container can be at the same time parent and child, we can build a hierarchical tree-like structure made of graphic elements. At the root of this tree we find the *main window*, which is the one that first appears in the computer screen or the one we embed within an html page.

There are, in turn, two main families of containers: containers which can hold basic elements and containers for drawables.

The first type is made of containers which can hold other containers and basic elements (see next subsection). Both sets, altogether, are used to create the squeleton of the user interface of the simulation.

The second type is made of specialized containers that can only host drawable elements (see subsection 6.1.3) as children. We'll refer to them in the appendices as *containers for drawables* (what else?☺). These containers and its children drawables are used to animate drawings or display graphs of the simulated phenomenon.

An important property of a container of the first family that we must also cover here is the *layout*. When a child element is added to a container, the parent gives the child a position and size according to the available space, the demands of other children and, so to say, to its own hosting policy. This is what we call the layout policy, or simply *layout*. Some layouts give the child the opportunity to choose a position within the parent, though in most others this position is a consequence of the "order of birth" of the child.

I recognise that, in a first approach, layouts usually seem a nuisance. However, they are actually very useful because they help control the appearance of the interface if the user changes the size of the main window, or the length or fonts of the texts (something that happens very often). When the parent gets its size changed, it takes care of resizing its children so that they fully cover their respective position. After a bit of practice, the use of layouts becomes very natural.

We shall only use the following basic layouts:

- *FlowLayout*, which places children in a row, from left to right, very similarly to how words are placed in a paragraph. Children can be left, center or right aligned.
- *BorderLayout* (perhaps the most popular), which places children in one of five positions: north, south, west, east and center, from which the children can choose.
- *GridLayout* (the second most popular) which places children in a rectangular grid with as many rows and columns as we specify.
- *Horizontal Box*. This works very much as a grid layout with one single row. However, differently to the grid, it doesn't force all its children to have the same size.
- *Vertical Box*. This is the vertical version of the previous one.

For these layouts, we can also specify the gap separating, horizontally or vertically, one child from the next one.

6.1.2 Basic elements

The group of basic elements is composed of a series of interface elements that can be used to decorate the view, to display and edit variables and also to trigger model actions.

These elements are very popular in most interactive programs and include labels, buttons, sliders, editing fields…

Basic elements can be added to containers (of the first family) but can not host other elements (that is, they are not containers themselves).

6.1.3 Drawables

The set of drawables is one of the main contributions of *Open Source Physics* project to **Ejs**. It consists of a series of view elements that can be included in dedicated containers (those of the second family described above) to display animated graphics that visualize the model states.

These animated graphics range from the simple to the very sophisticated, and include particles (represented as ellipses or rectangles), arrows, images, polylines, vector fields, contour plots, three dimensional bodies,… and a lot more.

6.1.4 Creating a View

All together, creating the view consists in generating a structure of elements that fits our needs. This structure must include:

a) elements that visualize the state of the model,
b) elements for user interaction with the simulation, and
c) containers that host all other elements.

6.2 Linking variables to properties

As we said before, graphic elements have certain internal values, called *properties* that can be customized to make the element behave in a particular way.

Because we are not only interested in creating nice computer graphics, but also in visualizing our model state using interactive interfaces, we want to use our model variables and actions as values for some of these elements properties. I will refer to this process as *linking* model items with view properties.

Really, if we just create a model and a view for our simulation, but we don't instruct the view about how to use model variables and actions, we could run our simulation, but our view would display nothing of interest. It is only when we establish the appropriate links that the view conveys useful information about our model.

In standard programming, these links are always created using predefined methods of the graphic components, which do what we want on the elements of the interface. The problem is that, since elements have been created for generic purposes, these methods are of a very low level.

Ejs dramatically simplifies this situation by having tailored all the elements that we will use under a linking scheme where all you need to do is to edit a panel with the properties of the element and indicating in each property field, which value you want to use for it. In most cases, you can do so this by choosing from a list of acceptable values the one that fits your needs.

Later, when running the simulation, **Ejs** automatically takes care of all the internal calls to Java methods needed for these links to work properly.

Linking is a two-way, dynamic process. This means that, at any moment, the property of the element will reflect whatever value the linked variable holds. And viceversa, if the property changes as a result of the interaction of the user with the view (such as typing in a new value, or moving a scrollbar), the variable of the model will receive the new value.

If an action is linked to an element *action* property, whenever the element is required to execute its action (for a button, for instance, when you press on it), the corresponding model action is called.

6.3 How a simulation runs

Once we have established the connections among the three parts of a simulation, we can complete the description of running a simulation that we started in section 5.1.3.

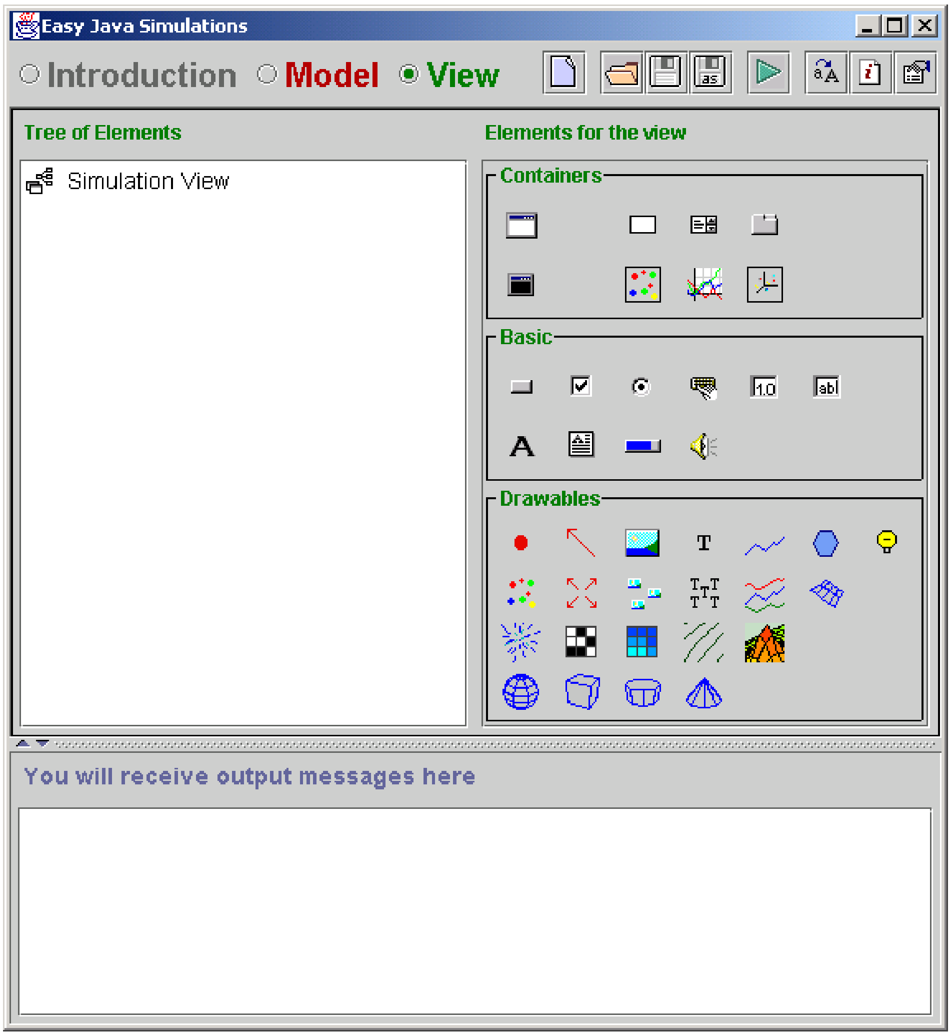
1. Variables are created and their values are set to those indicated in the initialization step.
2. The control and the view are created. The view appears on the screen.
3. Constraint equations are evaluated (since the initial value of some variables may depend on the initial values of other variables).
4. Control actions are linked to the actions of components of the view.
5. The model uses its connection to the view so that the latter displays the state of the model.

Now, the model is in its initial state, the view reflects it, and the simulation waits for an evolution step or for the user to interact with it.

6. In the first case, evolution equations are evaluated and, immediately after, constraint equations. We then reach the new state of the model in a new instant of time and the view is updated using the connection from the model.
7. In the second case, when the user changes a variable, constraint equations are evaluated and we obtain a new state of the model within the same instant of time. The view is updated using the connection from the model.

6.4 Building the view

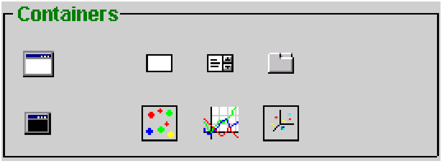
The interface of this part of **Ejs** looks as follows.



A panel that shows the (initially empty) structure of elements of the interface occupies the left-hand side of the working area of **Ejs**. When we add elements to this structure, this panel will be displaying them in a tree-like form.

The right-hand side of the working area contains three sets of icons, grouped by functionality, according to the categories described in section 6.1 . Each icon represents a class of elements that can be added to our view.

Elements in the first set of icons are used to add container elements to our view (containers were introduced in subsection 6.1.1).



The first class of elements that we need to mention from this group is the *Frame* class. It has the icon 🗔 and if you place the mouse over it and wait for a second, a small caption will appear that reads '*A top level window*'.

Every view needs at least an initial window in which we will place all other elements. The appropriate elements for this task are frames. They know how to handle all the interaction with your operating system windowed screen.

The icon ■, below the previous one, represents a slightly different class of basic windows, that of *Dialogs*. Dialogs are also windowed elements, but (unlike frames) they do not exit a simulation when they closed[29]. Hence, since you want to be able to somehow exit your simulation definitely, your simulation view needs to have at least one frame element.

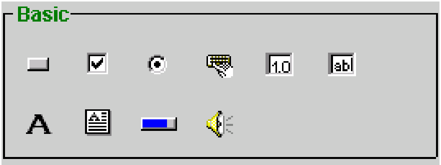
On its defense, dialogs have the ability to stay always visible on top of the frame that was created before them. This is very useful in multi-windowed simulations when you want to keep secondary windows (dialogs) always visible on top of the primary ones (frames).

Finally, and a bit separated from the previous two, a top row of icons represents other containers for basic elements (of which, without any doubt the most frequently used is the basic panel ▭), and a bottom row displays different classes of containers for drawables, both for 2D and 3D drawings.

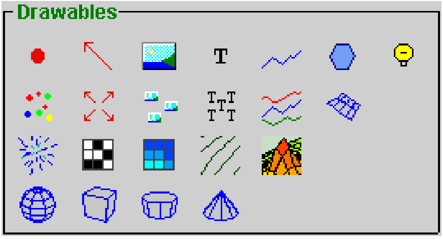
There are, at this moment, three such containers, *Drawing Panels*, *Plotting Panels* and *3D Drawing Panels*, their icons are ⁘, 📈 and ⊹, respectively.

Please notice that, although elements of these classes are containers, their only purpose is to hold just drawables. Therefore you cannot create basic elements as children of them. Also, since drawables draw in a position and size determined by the value of their internal properties, these containers panels have no *Layout* property.

The second set of icons holds elements that are used for basic operation. It features labels and buttons, which you can use to trigger actions, but also bars, sliders and fields, useful to display and modify model variables.



[29] Closed means closed, not minimized.



The final set of elements are of a special category called generically *Drawables*. They can be used, not to display the value of variables, but to perform certain drawings according to the value of their properties.

This is very convenient to make your visualization of the model state a bit more alive, and not just the cold display of figures.

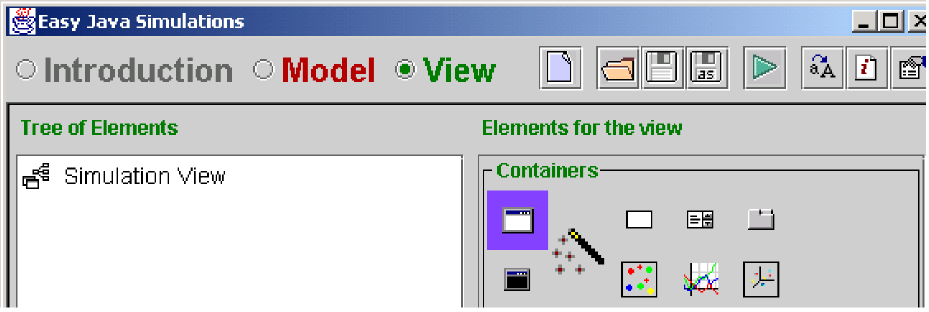
Let me say it once more, *Drawables* need to be added as children of the special containers described above that can handle their drawing peculiarities.

6.4.1 Adding elements to the view

The creation of new view elements follows next four steps.

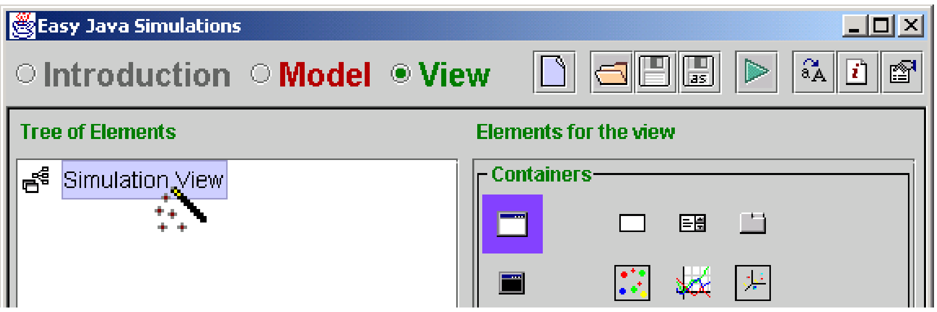
Step 1. Select the class of the element to be created

Just click on the icon which represent the class that you want to create. The background of the icon will change color, thus indicating that it is active. Also, the cursor, which is by default an arrow (that changes temporarily to a pointing hand when you are on top of an icon) will change into a magic wand, .

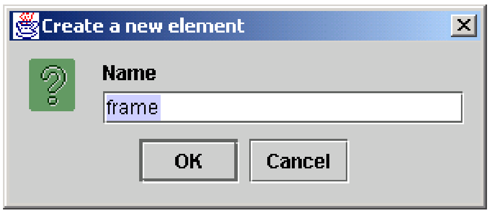


Step 2. Select the parent for the new element

For this, it suffices to click on a container component from the tree of components shown on the left panel. If you are creating a windowed element (that is, a frame or a dialog) which requires no parent, you have to click on *Simulation View*, the root node of the tree.



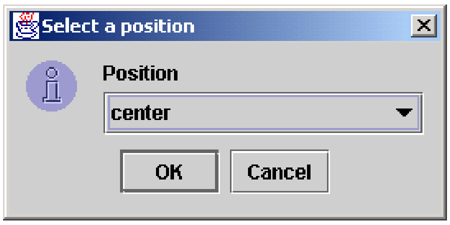
Step 3. Select a name for the new element



Name: frame

The new element needs a name. **Ejs** will propose you a name for the new element, but you can give it any other name (recall, however, naming conventions stated in subsection 5.3.3).

Step 4. If necessary, select a position for the new element

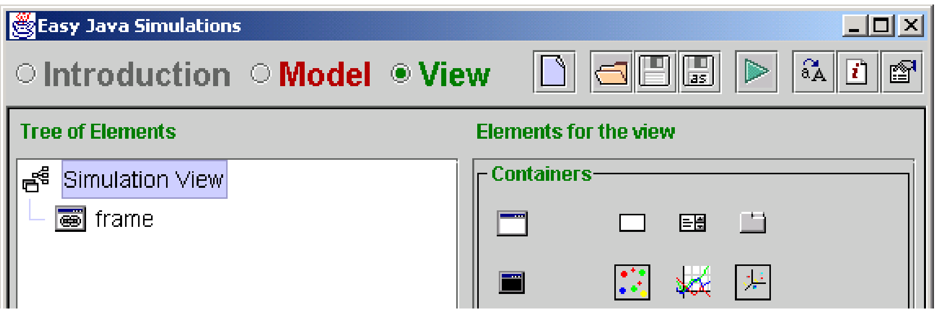


Position: center

Only if the layout of the selected parent is *BorderLayout*, you will be prompted to choose a position (*north, south, ...*) for the new element as child of it[30].

If the parent already has children, make sure not to choose a position that is occupied by a previous child. This could turn into strange configurations, with one child hiding another.

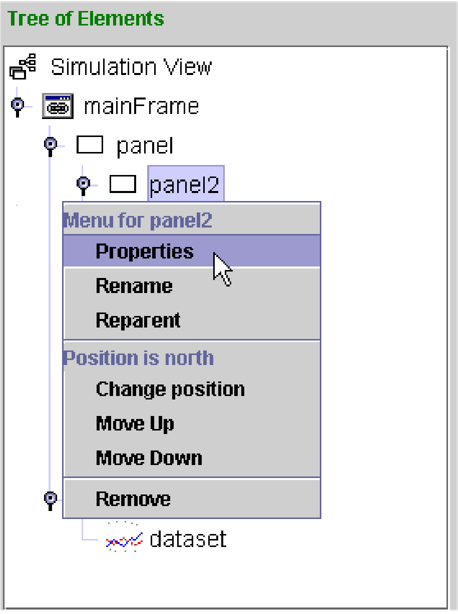
And this is all! The new element is created for you and allocated in the right place.



[30] I include the picture here to illustrate the step, but it does not corresponds to the same sequence that previous pictures show, since a frame that is created on the *Simulation View* root node does not need to choose a position.

6.4.2 Modifying the tree

Once we have created a tree by adding all the elements we need, or also while we are in the process of building it, we can modify the tree structure in different ways. This is sometimes necessary to correct any minor mistake that we might have incurred into, or simply to modify our initial configuration due to a change of mind or to new requirements.

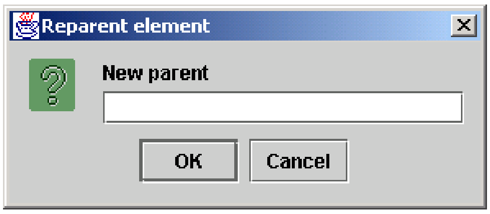


For this purpose, every element in the tree (except the root node) has a popup menu that we can bring in by right-clicking the mouse over it. The figure shows a simple view with the pop-up for one of its elements, *panel2*, on top.

This menu varies a little bit depending on the actual element that you select. But the picture shows a typical example.

The first entry in the menu allows you to bring in the edition dialog for the element properties. This, the most frequent option, will be described in the next section.

The second option allows you to change the name of the element, and the third can be used to change its parent.



In both of these cases, a dialog will appear to ask you for the new name, or for the new parent, respectively. The picture shows the dialog for reparenting.

In a second group in the menu, we find the options that help us change the position of the child within its parent. A label indicates us the current position of the element in the parent and, if the parent has border layout (as in the case shown in the picture), the first option allows us to change the position.

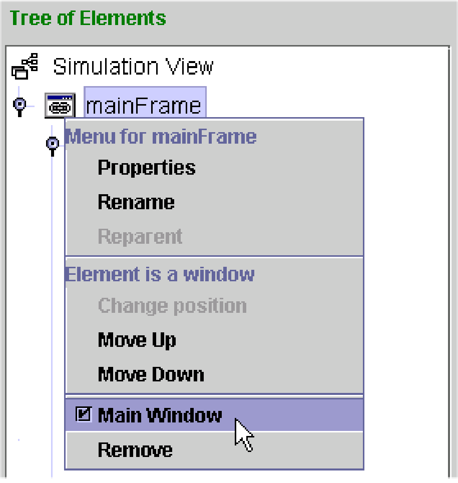
If the parent had a different layout, one that imposes the position on its children according to its order of creation, this option would be disabled. In this case, we can still change the position of the element by changing its relative order.

The last two options in this group let us change the position by moving the element up or down in the tree hierarchy (without leaving its parent).

A final option, which is separated from the rest, simply removes the element. If the element is a container, it will also remove all its children. Actually, the whole part of the tree that hangs from it (so, please use it with care!).

A SPECIAL OPTION, ONLY FOR FRAMES

A special option that appears only in menus for frames is the one labeled *Main Window*. Perhaps you noticed that while the icon for the *Frame* class is ▭, it appears as ▣ when you create your first frame. However, if you add more frames to your view, they will be displayed with the original icon. This has a special meaning.



When a simulation is run as an applet, within an html page (see chapter 4), you can choose which of your frames will be captured and form part of the html page itself. Only the *Main Window* will be captured, all other frames (also dialogs) will appear in separate windows.

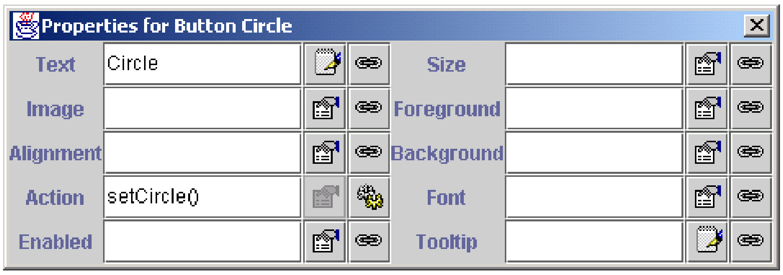
This option, that the picture shows selected for our *mainFrame*, allows you to select which of your frames will be your main window.

6.5 Editing properties

When we create new elements, they appear with default values for their properties. As said before, properties are internal values that determine how the element behaves and looks on the screen.

We can modify these properties using the edition dialog for the properties of the element. We do this by selecting the first option of the element popup menu, as described in previous subsection.

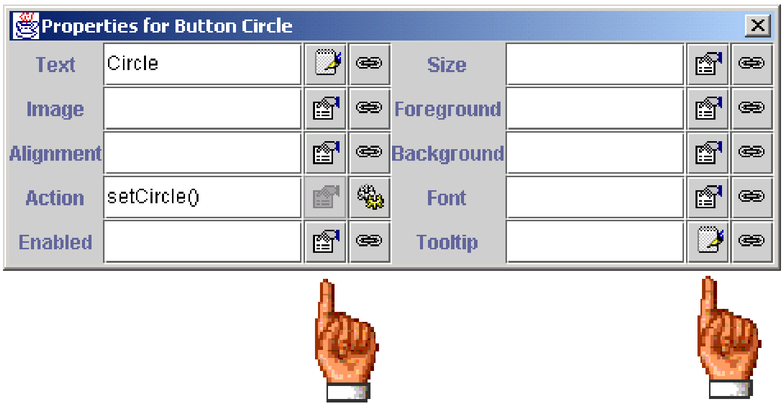
The picture below shows the edition dialog for an element of the class *Button*.



Text	Circle	Size	
Image		Foreground	
Alignment		Background	
Action	setCircle()	Font	
Enabled		Tooltip	

The way to modify a property is rather straight-forward. You click in the corresponding field and type in the value you want to give to the property.

Sometimes, however, it is not so easy to remember the right format for some of these values, specially in cases where the property is a technical one, like color, layout, and font, for instance. For this reason, the text fields have a first utility button (we'll talk about the second utility button in a minute) with a descriptive icon to its right.

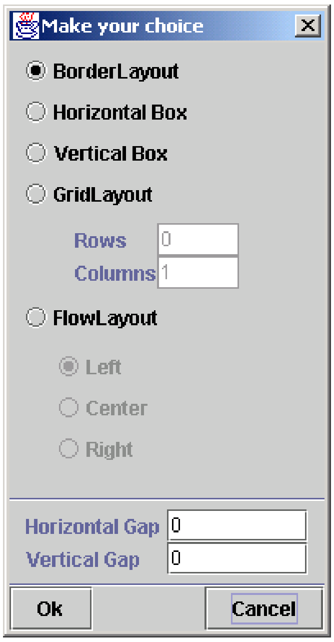


Text	Circle	Size	
Image		Foreground	
Alignment		Background	
Action	setCircle()	Font	
Enabled		Tooltip	

If the icon is this one 📝, then there is no help to edit this field: you need to type in the value that you want. Usually in this case, the property is a simple one, like the text that appears on the buttons or labels or on the titles of frame or just numerical constant values.

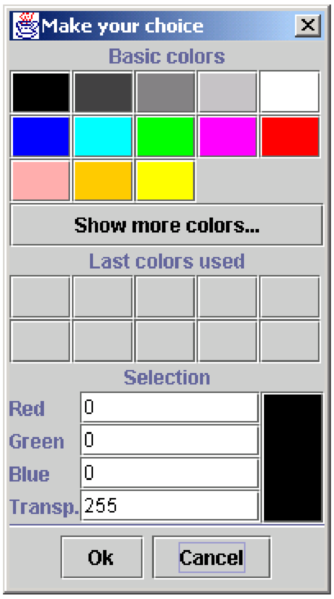
If the icon is the following 📋, then this implies that **Ejs** has a dedicated special editor to help you choose a valid value for it. I recommend that you always use this editor when offered to.

Pictures below show the dedicated editors for layout, color and font, respectively. They all work in a pick and choose manner.

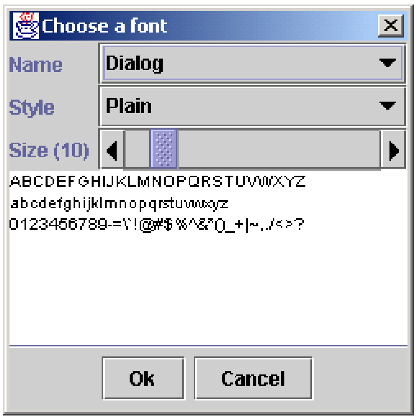

Make your choice

● BorderLayout
○ Horizontal Box
○ Vertical Box
○ GridLayout
Rows 0
Columns 1
○ FlowLayout
● Left
○ Center
○ Right

Horizontal Gap 0
Vertical Gap 0


Make your choice

Red 0
Green 0
Blue 0
Transp. 255


Choose a font

Name Dialog
Style Plain
Size (10)

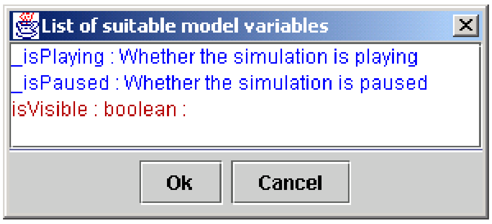
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
0123456789-=\!@#$%^&*()_+|~,./<>?

Finally, if the button is disabled, then this property can only be linked to a model variable or action (see below).

There is a second column of utility buttons, one for each property field, that show either the icon 🔗 or ⚙. Recall that properties can also be linked to variables from the model; this button will help you do just this. Linking properties to model variables is at the heart of dynamic interactive interfaces.

If you click on a button with the link icon, 🔗, a dialog with a list of all model variables that can be chosen appears. Since the property you are editing may require that the variable you associate to it is of a special type (i. e., *boolean*, *int*,…) only model variables of the corresponding type appear in this dialog. To select one of the variables, click on it and then on *Ok*.

For instance, if we click on the button to the right of the *Enabled* property of the button, we will get a list of all boolean variables of our model.

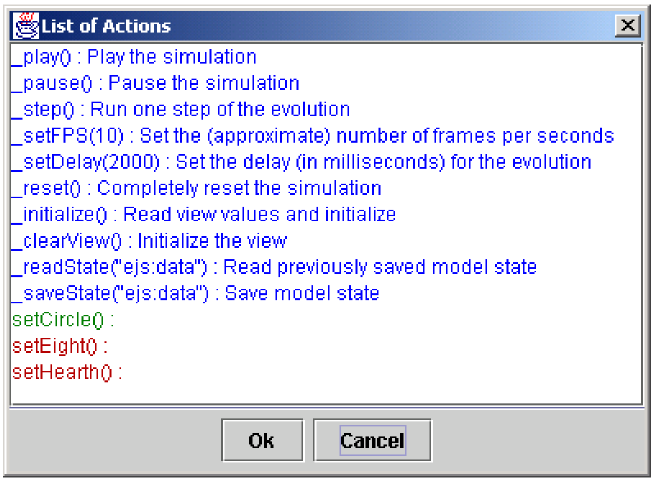

List of suitable model variables

_isPlaying : Whether the simulation is playing
_isPaused : Whether the simulation is paused
isVisible : boolean :

The list in the picture above assumes that your model has declared a boolean variable called *isVisible*. Besides the boolean variables of the model itself, the list also includes, in case they are of use for you, the two boolean variables provided by the system, namely *_isPlaying* and *_isPaused* which are true if the model is playing and paused, respectively. These are displayed in blue color, while model variables are displayed in the model family color, red. Your choice will be displayed in green.

Finally, if this second button shows the action icon, ⚙, this means that the property belongs to the category of *actions*. That is, that you can link this property with a model action, or with one of **Ejs**' predefined actions, so that, when you interact with the element, in a form that depends on the element and on the particular action property, the corresponding action will be called.

If we click on the ⚙ button, a dialog listing all the actions that you can choose will appear.



_play() : Play the simulation
_pause() : Pause the simulation
_step() : Run one step of the evolution
_setFPS(10) : Set the (approximate) number of frames per seconds
_setDelay(2000) : Set the delay (in milliseconds) for the evolution
_reset() : Completely reset the simulation
_initialize() : Read view values and initialize
_clearView() : Initialize the view
_readState("ejs:data") : Read previously saved model state
_saveState("ejs:data") : Save model state
setCircle() :
setEight() :
setHearth() :

Again, the list displays in blue color **Ejs**' predefined actions and in red (model's favourite color) model actions.

SPECIAL CASE 1: PROPERTIES OF TYPE *STRING*

You may have notice that there might be an ambiguity in the case of a property which accepts a String as its value.

For instance, if we edit the property *Title* of a frame and give to it the value *main* : does this mean that the frame should display the word *'main'* or rather that *main* is a variable and that the frame should display whatever value *main* holds?

In most cases, the property decides what is the option most frequently used and will take this as the desired result. However, there is a way to clearly solve this ambiguity:

- typing the constant String value either between quotes or inverted commas will force the element to take this entry as a constant value. In our example, write *'main'* or *"main"*

- typing the variable between percent characters *%*, will force the elementy to consider this a variable. In our example, write *%main%*

In any case, this ambiguity can occur if you type directly the value for the property. If you use the dedicated editors, the system will include the necessary distinction tags.

SPECIAL CASE 2: PROPERTY *SIZE* OF WINDOWS

Windowed elements, that is, frames and dialogs, show a special behavior that allows to modify their size in an interactive form. When editing their *Size* property, i.e. their size on the screen, you can follow any of the previous two procedures or, additionally, resize the window on the screen in the way you normally use to resize windows in your operating system environment. When you resize the window, the actual values are reflected automatically on the field for the *Size* property.

6.6 Learning more about view elements

And this is, basically, all you need to know about view elements!

Of course, you still need to master all the elements that you can use within **Ejs**. But for this, you'll need to refer to the appendices. You will find there a description of each class of element, together with information about their properties and how to modify them.

We will close this manual creating a second complete simulation. Please read it completely, because there are some other, though minor, things to learn that you will only find referenced there.

I also recommend, once more, that you don't forget to have a look at the examples bundled with **Ejs**. I think you will find lots of tricks and ideas for your own simulations in them.

This page intentionally left blank

7. A second example: predator-prey systems

Let us create a second complete example to illustrate some other aspects of the process of writing a simulation. For this we choose an interesting phenomenon from the domain of population dynamics: the well-known dynamical system of a predator and its prey.

7.1 Description of the phenomenon[31]

During the middle 20's (1920's), Italian biologist Humberto d'Ancona discovered, studying the volume of different species of fishes captured in the Mediterranean Sea during the First World War, that the percentage of *selacii* (sharks, rays, dogfishes), a type of fish, in general, not appropriated for human consumption, had increased significantly.

Obviously, the number of total captures had decreased during the war and, consequently, the number of fishes in the sea should be larger, but this did not explain why the reduced level of fishing was benefiting, in relative terms, selacii versus other fishes.

A possible explanation was that selacii are predators that eat, among others, the same fishes as humans. Hence, when the fishing decreased, the volume of preys increased and selacii grew quickly in number. However, this argument is not consistent since lower levels of fishing should have also benefited the preys, in a similar percentage.

The problem came then to the hands of the reputed Italian mathematician Vito Volterra, who proposed the following model (also studied independently by Lotka).

Call *x(t)* and *y(t)* the populations of preys and predators, respectively, at instant *t*. Since fishes for human consumption do not strongly compete among

[31] This description of the problem has been taken, with permission of the author, from the monograph **Ecuaciones Diferenciales: cómo aprenderlas, cómo enseñarlas** written by Dr. D. Víctor Jiménez López, Universidad de Murcia, 1999.

themselves for food, because it is abundant, their number grow, in the absence of predators, according to the Malthusian law *x' = ax*, for some constant *a>0.* On the other hand, the number of contacts between predators and preys (which usually produce a fatal result for the prey), per time unit, can be written as *bxy*. We therefore obtain *x' = ax – bxy*.

Similarly, the number of predators increases according to their population *y* and available food *x*, and decreases according to their death rate. Putting everything together, we obtain:

$$\begin{cases} x' = ax - bxy \\ y' = -cy + dxy \end{cases}$$

for positive appropriate constants *a, b, c* and *d*.

This is then our initial model for the simulation. We are interested in the evolution of the populations and of the average number of predators and preys along time.

It is surprising to check that these averages do not depend on initial conditions, that is, on the initial number of fishes of both types (except the trivial case when one of them is zero).

Furthermore, if we now introduce in the model the effect of fishing, that is, an amount proportional to both populations *ex* and *ey*, we get our final model:

$$\begin{cases} x' = (a - e)x - bxy \\ y' = -(c + e)y + dxy \end{cases}$$

It is even more surprising to check that a moderated level of fishing (*e<a*) has the effect of increasing (in average) the volume of fishes for human consume and decreasing the number of selacii. If the level of fishing is small (e<<a), we observe the opposite effect.

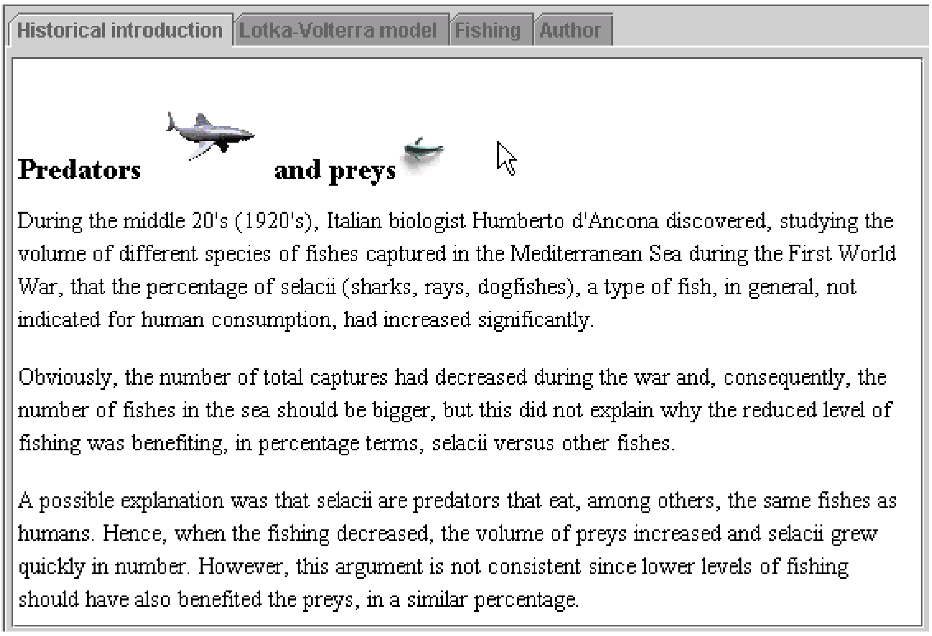
This surprising result, known as *Volterra's Principle*, explains satisfactorily the observations of D'Ancona.

We will first create a simple simulation that implements this model and displays the evolution of the populations. In a second step, we will improve the simulation to let the student experiment with it: changing initial conditions, playing with the parameters and observing the result of her actions on the average number of predators and preys.

7.2 Introduction

We could create one or more introductory pages and write in them the introduction given above for this problem. Since doing this doesn't imply any particular difficulty (but having the time to type all this text in), I suggest that you take a look at the finished simulation file included in the example *PredatorAndPreyBasic* distributed with **Ejs**, to see how this finally looks.

The picture below shows the first page of the introduction for this example.



Predators and preys

During the middle 20's (1920's), Italian biologist Humberto d'Ancona discovered, studying the volume of different species of fishes captured in the Mediterranean Sea during the First World War, that the percentage of selacii (sharks, rays, dogfishes), a type of fish, in general, not indicated for human consumption, had increased significantly.

Obviously, the number of total captures had decreased during the war and, consequently, the number of fishes in the sea should be bigger, but this did not explain why the reduced level of fishing was benefiting, in percentage terms, selacii versus other fishes.

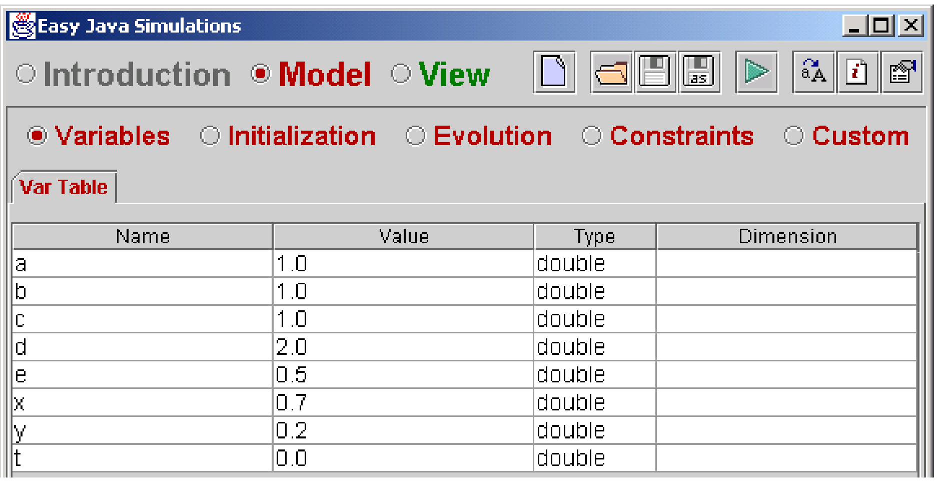
A possible explanation was that selacii are predators that eat, among others, the same fishes as humans. Hence, when the fishing decreased, the volume of preys increased and selacii grew quickly in number. However, this argument is not consistent since lower levels of fishing should have also benefited the preys, in a similar percentage.

7.3 Model

7.3.1 Step 1: Define the variables

From the equations above, we easily identify the state variables *x* and *y*, and the parameters *a*, *b*, *c*, *d* and *e*. They are all real numbers; hence we will use variables of type *double*. Though not explicitly mentioned, time is another variable, which we simply name *t*.

It then suffices to create the following table of variables:


Easy Java Simulations
○ Introduction ● Model ○ View
● Variables ○ Initialization ○ Evolution ○ Constraints ○ Custom
Var Table

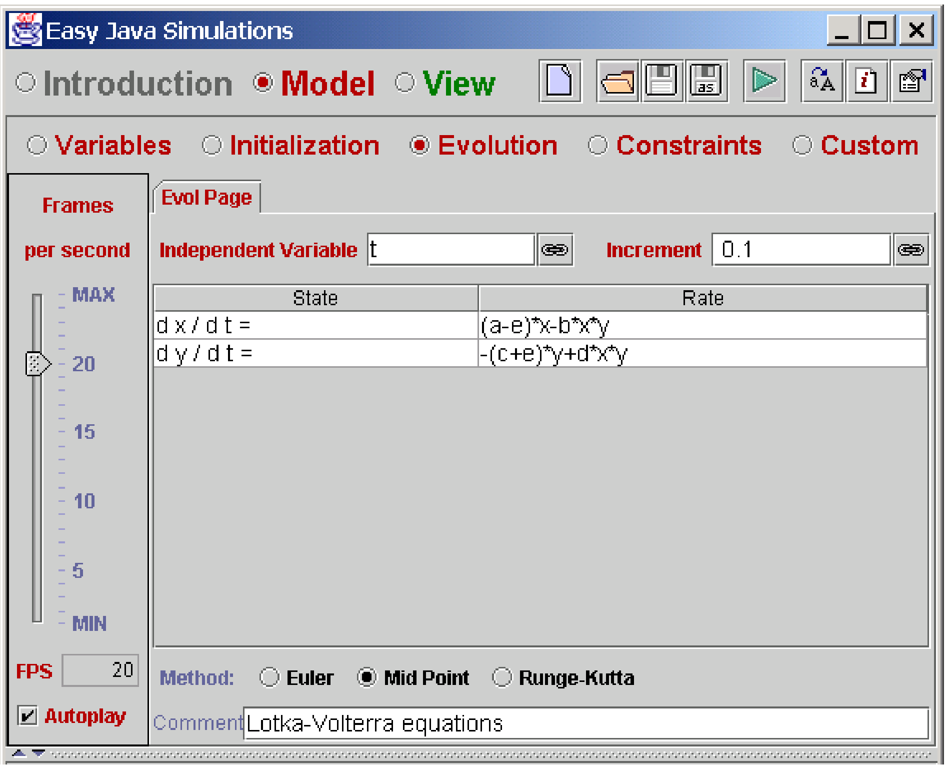
Name	Value	Type	Dimension
a	1.0	double	
b	1.0	double	
c	1.0	double	
d	2.0	double	
e	0.5	double	
x	0.7	double	
y	0.2	double	
t	0.0	double	

7.3.2 Step 2 : Initialize the variables

The initialization of the model is completed giving initial constant values to all variables, which we have already done in the previous step. This panel remains, therefore, empty.

7.3.3 Step 3 : Write evolution equations

This step is going to be straightforward, thanks to **Ejs**' ODE editor.


Easy Java Simulations
○ Introduction ● Model ○ View
○ Variables ○ Initialization ● Evolution ○ Constraints ○ Custom
Frames
per second
MAX
20
15
10
5
MIN
Evol Page

FPS 20

☑ Autoplay

Independent Variable t

Increment 0.1

State	Rate
d x / d t =	(a-e)*x-b*x*y
d y / d t =	-(c+e)*y+d*x*y

Method: ○ Euler ● Mid Point ○ Runge-Kutta

Comment Lotka-Volterra equations

Notice that we have chosen to start automatically the evolution and a frames per second rate of 20, which implies a small delay in the execution of the simulation. We have also selected to apply the mid-point algorithm for the numerical resolution of the differential equations.

7.3.4 Step 4 : Write constraint equations

Our model needs no constraint equations.

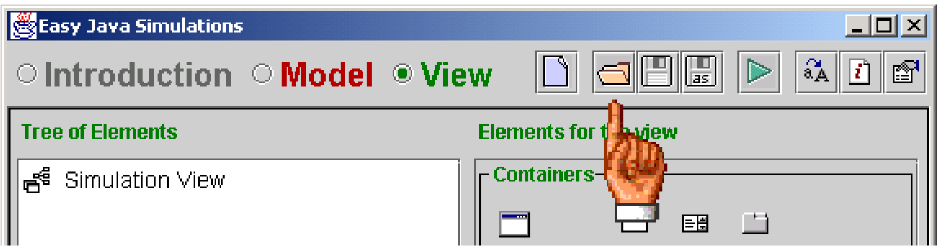
7.3.5 Running the model?

The model for our simulation is now fully specified. If we ask **Ejs** to run the simulation, we will have our model running and, approximately 20 times per second, the differential equation will be solved and the values of x and y changed according to the evolution in time of our two populations.

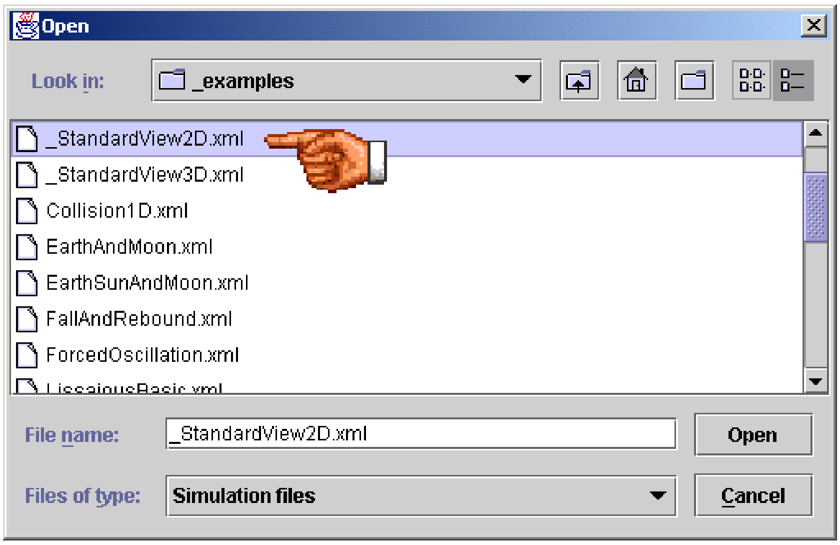
However, we will see nothing because we haven't created a view for our model, and we will not be able to control the simulation (stop it, for instance).

7.4 View

For the view we will reuse a generic interface included with **Ejs** distribution. I we click on the *Open an existing file* button on the top toolbar,



and move to the *examples* directory, we will find the file called *_StandardView2D.xml*.



Look in: _examples

_StandardView2D.xml
_StandardView3D.xml
Collision1D.xml
EarthAndMoon.xml
EarthSunAndMoon.xml
FallAndRebound.xml
ForcedOscillation.xml
LissajousBasic.xml

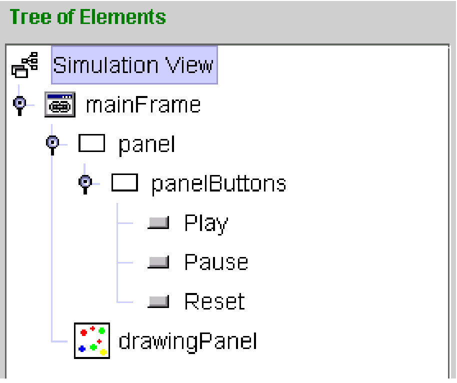
File name: _StandardView2D.xml

Files of type: Simulation files

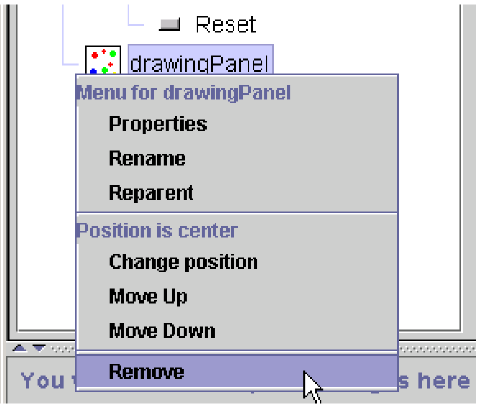
Files with a name that starts with an underscore character, like this one, have a special characteristic: they are not read, but *merged*. This means that they will not clear **Ejs** editors before reading whatever is in the file.

> Since the only requisite for creating a merge file is to give it a name that starts by '_', you can also create your own favorite standard views. This can be of particular use if you (like me) have a tendency to use frequently the same kind of view configuration.

Now, please merge this file and notice that this adds some elements to our empty view.



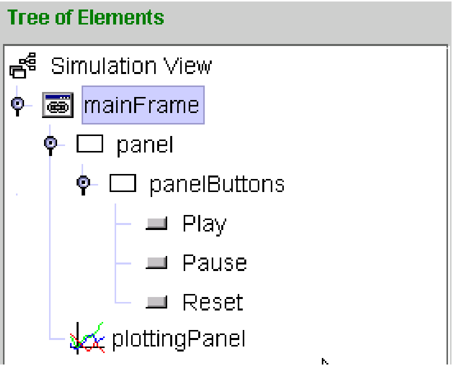
The new view consists of a main frame with a simple Play-Pause-Reset control panel to its left and a central drawing panel. However, we will prefer to change the drawing panel for a plotting panel, since we want the axis to be automatically plotted, too.



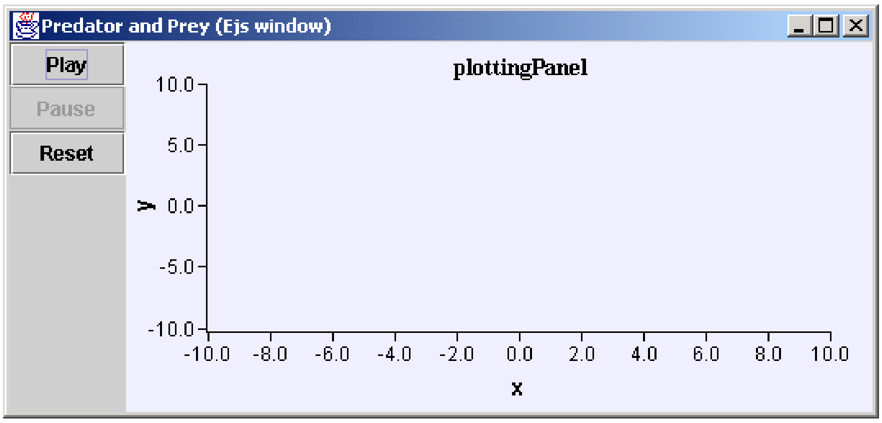
To achieve this, we first remove the element drawingPanel, …

and then add a new plotting panel element, the one with the icon , in the central position of *mainFrame*.

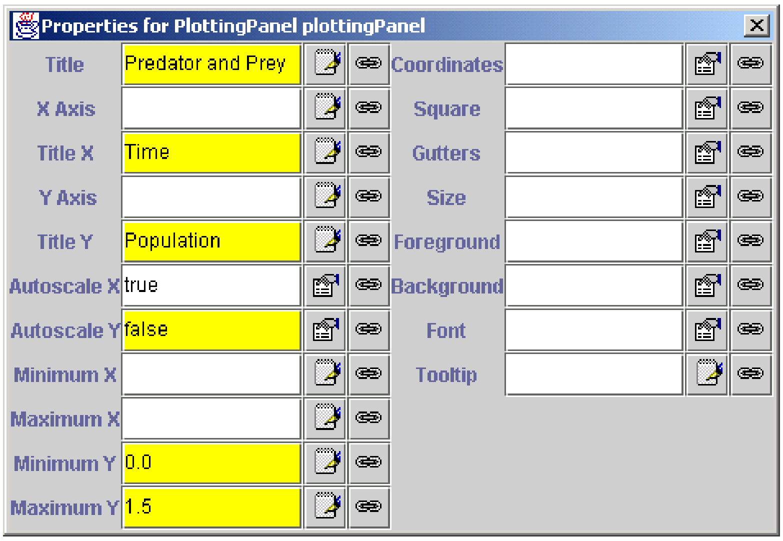
Now our view tree looks like this.



After modifying *mainFrame*'s title property to *Predator and Prey* and its size, this is how our view looks like.



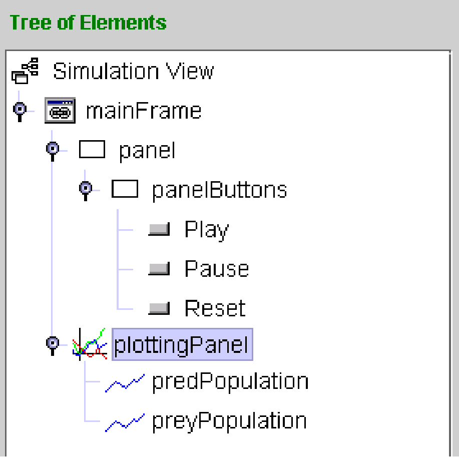
We will also edit the properties of *plottingPanel* in order to change the titles and adjust the scales. Next picture shows the options we will take (I have made the changes visible by changing the background color of the corresponding fields, but please recall to hit return after typing the entries).


Properties for PlottingPanel plottingPanel

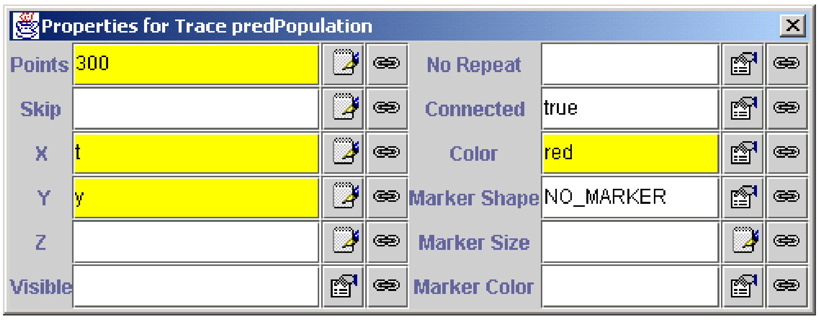
Title	Predator and Prey	Coordinates	
X Axis		Square	
Title X	Time	Gutters	
Y Axis		Size	
Title Y	Population	Foreground	
Autoscale X	true	Background	
Autoscale Y	false	Font	
Minimum X		Tooltip	
Maximum X			
Minimum Y	0.0		
Maximum Y	1.5		

To our view, we will now add two drawables that will display the evolution of the species by plotting the sequence of values *(t,x)* and *(t,y)*.

We do this by clicking on the icon for the *Trace* class (its caption reads *A sequence of points*) to select it, and hitting with the magic wand on our *plottingPanel*. If we give the new elements the names *predPopulation* and *preyPopulation*, this is the resulting tree.

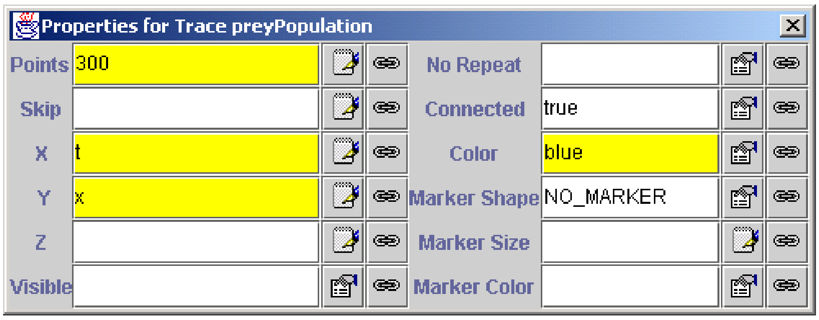


Finally, we edit the properties for these two trace element so that they can visualize our model variables appropriately.



Properties for Trace predPopulation

Points	300	No Repeat	
Skip		Connected	true
X	t	Color	red
Y	y	Marker Shape	NO_MARKER
Z		Marker Size	
Visible		Marker Color	



Properties for Trace preyPopulation

Points	300	No Repeat	
Skip		Connected	true
X	t	Color	blue
Y	x	Marker Shape	NO_MARKER
Z		Marker Size	
Visible		Marker Color	

This is all we need to do. We are taking from the model the variables *t*, *x* and *y* and telling the view to display the pairs *(t,x)* and *(t,y),* up to a maximum of *300* points. The *Autoscale X* feature of *plottingPanel* will make it appear as a stripchart recorder.

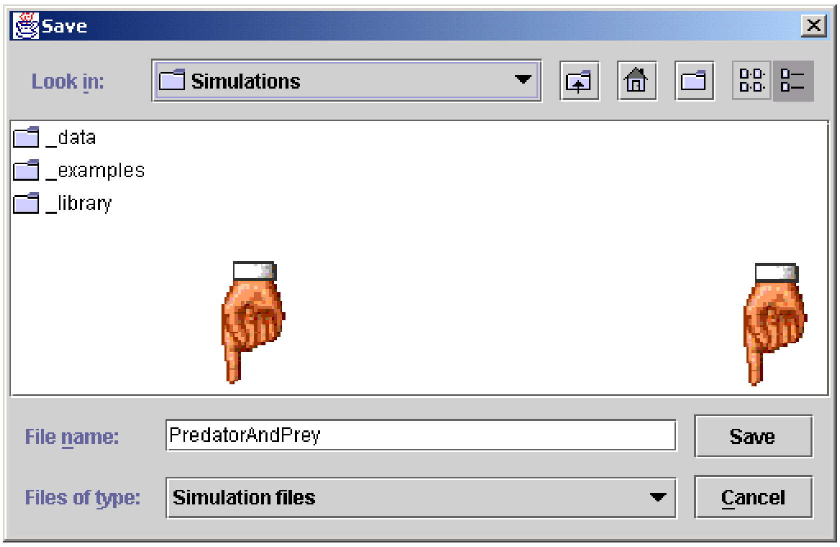
We don't need to link the *Action* property of the buttons to the predefined actions *_play()*, *_pause()* and *_reset(),* because this was already done in the file that we merged in. You can inspect, if you want, the property edition dialog for these elements.

7.5 Running the simulation

We can now run our simulation. But, before doing so, we should give it a name[32].

Click on the *Save* icon 💾 on **Ejs**' top toolbar, move to your **Ejs** home directory by cliking on the 🏠 icon of the file dialog, and then type a descriptive name for our simulation, say, *PredatorAndPrey*. Then, click on *Save*.

[32] If we didn't give it a name, **Ejs** would use the default name *Unnamed* to generate our simulation, but it would not save it.



Save

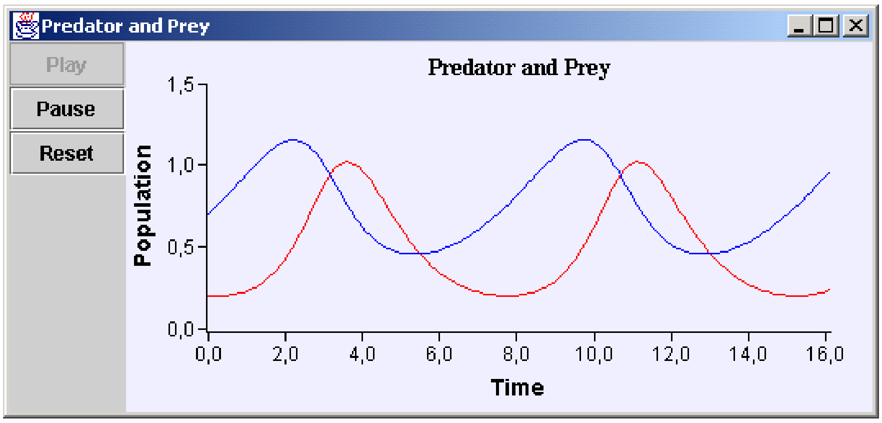
Look in: Simulations

_data
_examples
_library

File name: PredatorAndPrey

Files of type: Simulation files

You can now run the simulation, just click on ▶. A typical execution produces the following picture.



Let me say, once more, that a given model can be linked to different views. For instance, in our example, we could have chosen to display the phase diagram given by the sequence of points *(x,y)*, instead of the time diagrams *(t,x)* and *(t,y)*.

7.6 Improving the use of our simulation

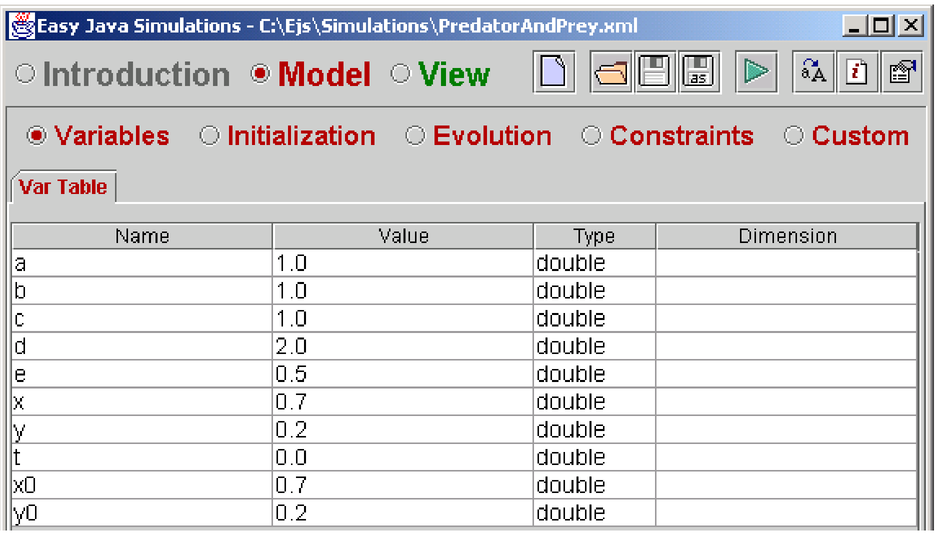
Even when it solves the model correctly and the view displays the evolution of the populations, the didactical use of our simulation is very limited because of the fact that it permits almost no user interaction.

What we really want is to let the student set different initial conditions for the model and check, if only visually, that the average of preys remains the same. Also, we want her to realize Volterra's Principle, that is, let her increase the level of fishing and obtain a higher average of fishes for human consumption.

Of course, the student could use directly **Ejs** and modify the initial values on the table of variables and run once and again. There is however a second approach that consists in giving the final simulation a reasonable amount of interaction. We will do this now.

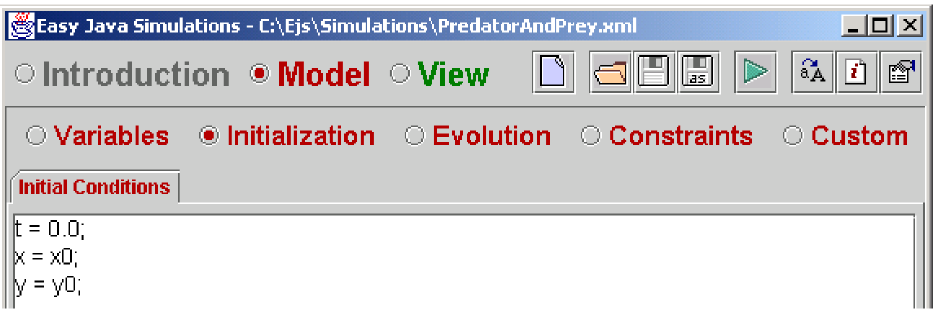
7.6.1 Changes to the model

Go back to the variables table and add two more *double* variables, *x0* and *y0*, our initial conditions, and give them the values *0.7* and *0.2*, respectively.


Easy Java Simulations - C:\Ejs\Simulations\PredatorAndPrey.xml
○ Introduction ● Model ○ View
● Variables ○ Initialization ○ Evolution ○ Constraints ○ Custom
Var Table

Name	Value	Type	Dimension
a	1.0	double	
b	1.0	double	
c	1.0	double	
d	2.0	double	
e	0.5	double	
x	0.7	double	
y	0.2	double	
t	0.0	double	
x0	0.7	double	
y0	0.2	double	

Now, in the *Initialization* panel, create a new page, say *Initial Conditions,* and type in the following code:


Easy Java Simulations - C:\Ejs\Simulations\PredatorAndPrey.xml
○ Introduction ● Model ○ View
○ Variables ● Initialization ○ Evolution ○ Constraints ○ Custom
Initial Conditions

```
t = 0.0;
x = x0;
y = y0;
```

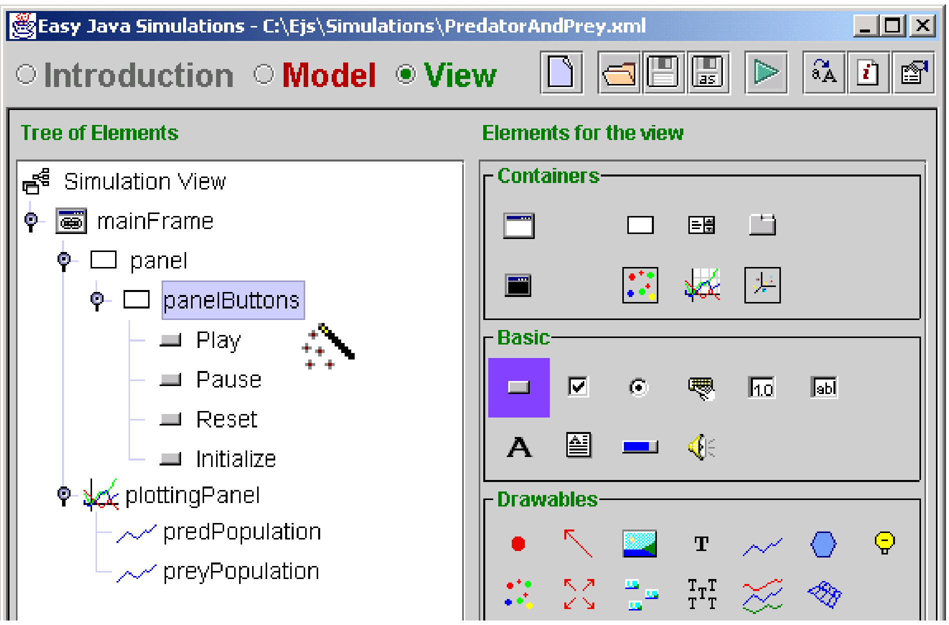
The effect of this code is to initialize our state variables *t*, *x* and *y* to the initial conditions *t=0*, *x(0)=x0* and *y(0)=y0*.

> You may think that this is not necessary since, after taking the initial values from the table of variables, these variables have precisely these values. Well, you are right, but… what would happen if we called *_initialize()*, instead of *_reset()*?… Please wait a little bit for an explanation.

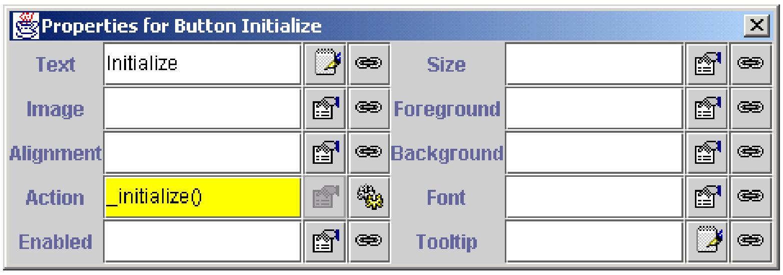
We don't need any further change to the model.

7.6.2 Changes to the view

First of all, we want to add a new button that will call *_initialize()*. For this, select the button class and hit with the magic wand on *panelButtons*. Give the new button element the name *Initialize*.

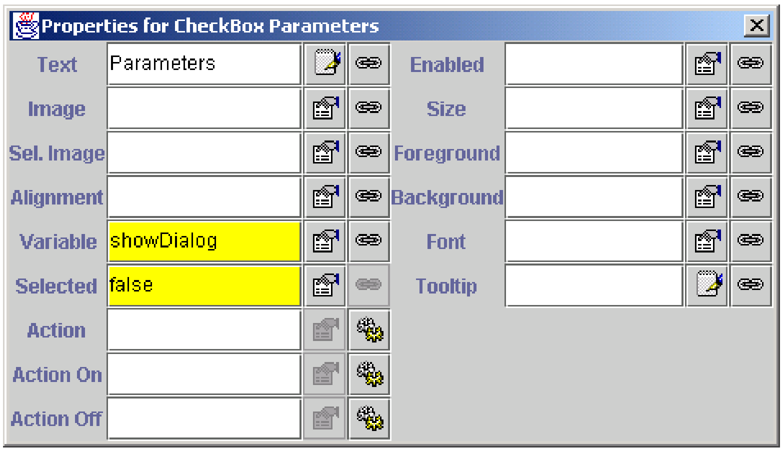


Since we gave it a name which also serves for the text for this button, we only need to edit its *Action* property and link it with the predefined method *_initialize()*.


Properties for Button Initialize

Text	Initialize	Size	
Image		Foreground	
Alignment		Background	
Action	_initialize()	Font	
Enabled		Tooltip	

We also add to *panelButtons* a checkbox ☑ which we name *Parameters* and edit its properties as follows.

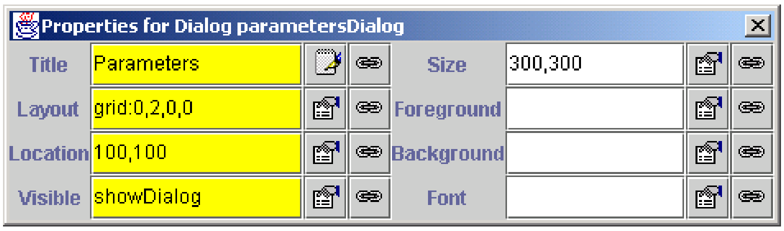


Properties for CheckBox Parameters

Text	Parameters	Enabled	
Image		Size	
Sel. Image		Foreground	
Alignment		Background	
Variable	showDialog	Font	
Selected	false	Tooltip	
Action			
Action On			
Action Off			

You could object that we have no variable in the model with the name *showDialog*. And you would be right. What this example tell us is that the view can have its own variables, that don't even need to be declared beforehand. If they happen to have the same name as an existing model variable, then they are linked, if they don't, the view assumes that you want to create a new view variable. We'll see the use of this soon.

We now add a dialog ▣ window to the view (hit the magic wand on the *SimulationView* node), with name *parametersDialog*, and edit its properties to look like this.



Properties for Dialog parametersDialog

Title	Parameters	Size	300,300
Layout	grid:0,2,0,0	Foreground	
Location	100,100	Background	
Visible	showDialog	Font	

Please take special care to type *showDialog* exactly as you typed it before. Any change in case would make it different.

Now, you can see what is the use of having internal variables in the view.

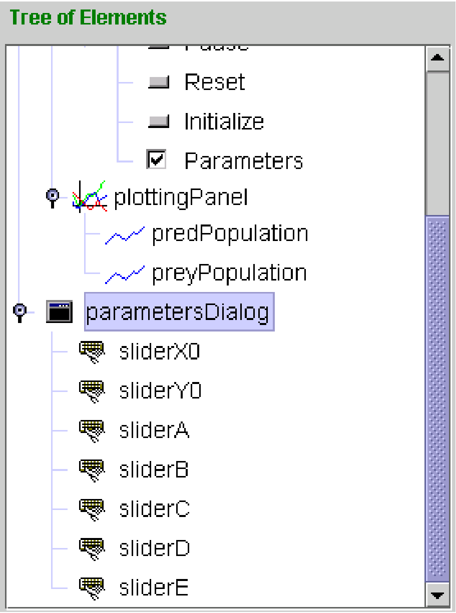
Even if there exist no model variable counterpart, the view can share information among its different elements using its own internal variables. You can check that now, if you select and unselect the *Parameters* checkbox, the *parametersDialog* appears and dissapears from the screen. Also, if the dialog is visible and you close it (in the way you close windows in your operating system) the checkbox is unselected.

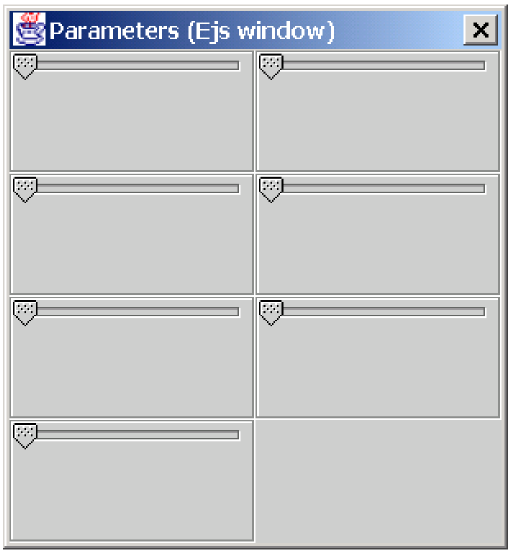
Although you could have used a model variable for this, there is really no need to declare one if you are not going to modify it in the model's logic.

Finally, we add view elements that will let the user modify the parameters of the model (i.e. , *x0*, *y0, a, b, c, d* and *e*) by interacting with the view. One of my favorite element classes for doing this is the *Slider* class .

We need to add six of them to *parametersDialog*. Since the dialog has *GridLayout*, we don't need to specify the position of each; the parent will be placing them according to the order in which you create them.

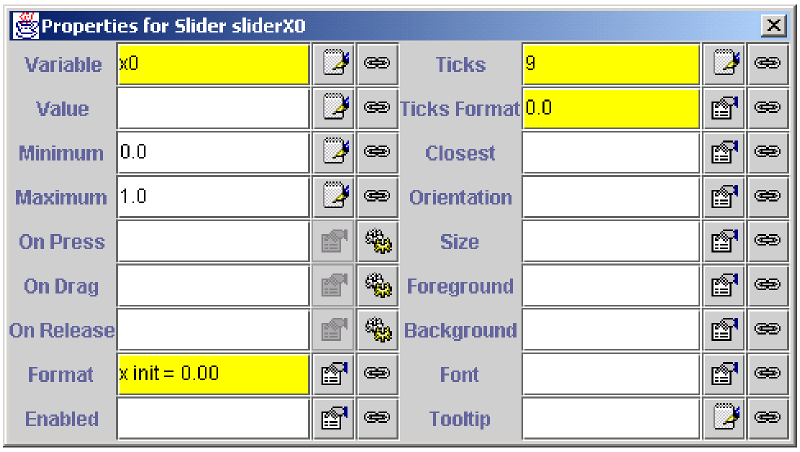
Give them the names *sliderX0*, *sliderY0*, *sliderA*,… and so on. This is how the final tree and *parametersDialog* look like for me.





Ok, I agree. The sliders don't look very nice, right now. Please wait, we need to customize them so that they properly display the value of our variables.

Edit the properties of *sliderX0* in the following way.

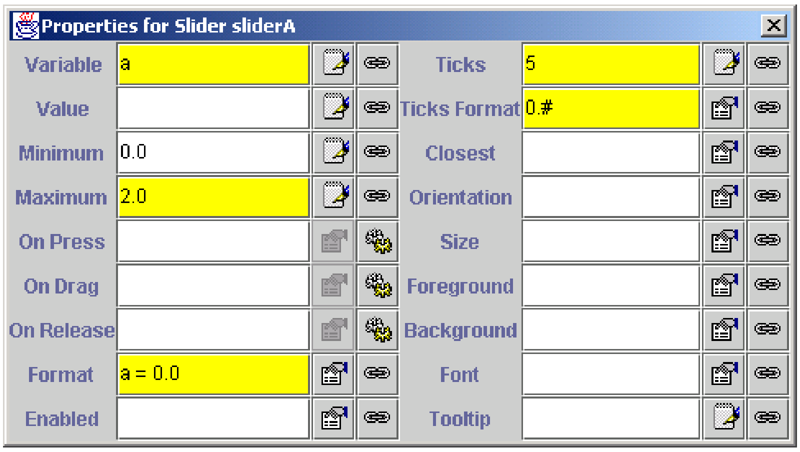


Properties for Slider sliderX0

Variable	x0	Ticks	9
Value		Ticks Format	0.0
Minimum	0.0	Closest	
Maximum	1.0	Orientation	
On Press		Size	
On Drag		Foreground	
On Release		Background	
Format	x init = 0.00	Font	
Enabled		Tooltip	

For the *sliderY0* use the same values for the properties, but using *y0* instead of *x0*, and *y init = 0.00* instead of *x init = 0.00*.

Now, for each of the sliders for the other parameters, edit their properties to look like this.

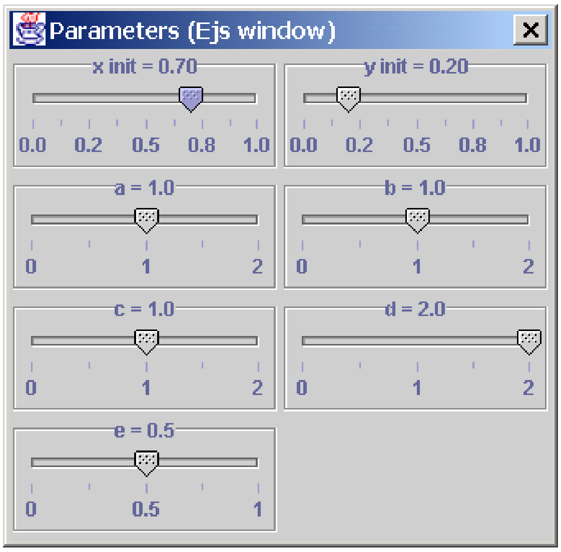


Properties for Slider sliderA

Variable	a	Ticks	5
Value		Ticks Format	0.#
Minimum	0.0	Closest	
Maximum	2.0	Orientation	
On Press		Size	
On Drag		Foreground	
On Release		Background	
Format	a = 0.0	Font	
Enabled		Tooltip	

Of course, this is for *sliderA*. For the others, replace a with the name of the corresponding parameter (exceptionally, for *sliderE* leave *Maximum* at *1.0*).

The final look of *parametersDialog* should be similar to the one shown in the next picture. You can notice that (they look much nicer and) the sliders display the actual value, as given in the table of variables, of the corresponding parameter.



x init = 0.70

y init = 0.20

a = 1.0

b = 1.0

c = 1.0

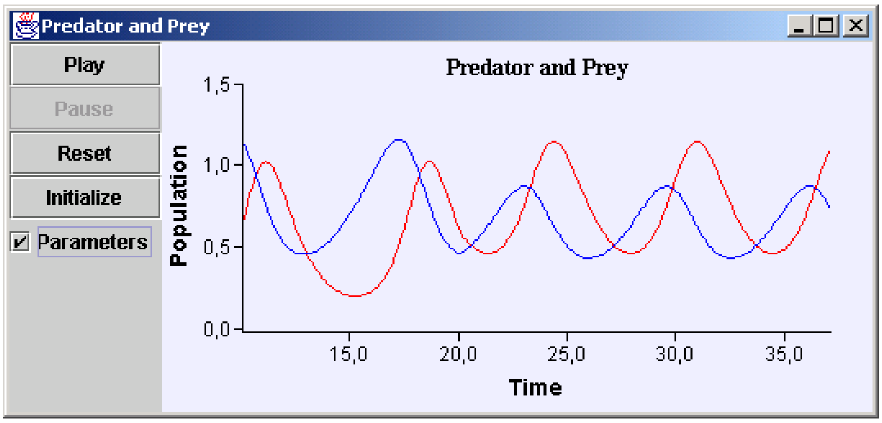
d = 2.0

e = 0.5

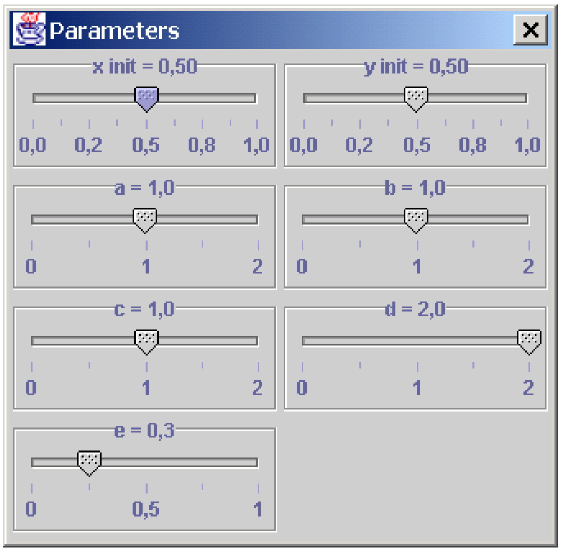
7.6.3 Running the simulation again

If you now run the new version of the simulation, you will see that it runs pretty much like the previous one. However, if you click on the *Parameters* checkbox, the dialog with the sliders appears.

You can modify the values for the parameters and see that the evolution of the populations change. I reduced the fishing to about half of its initial value at (approximately) *t=20.0* and this is what I obtained.



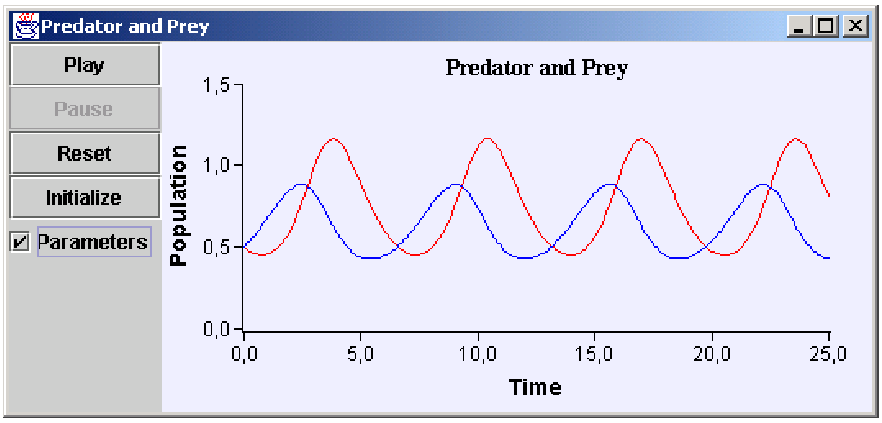
Can you appreciate what happens to the relative averages of both populations?… Volterra's Principle!



If you now click on *Reset*, the simulation changes back to it exact initial state.

However, if you modify the parameters and click on *Initialize* you will notice that the time is reset to zero and *x* and *y* take the values of *x0* and *y0*, but the value of the parameters that you edited using the view, including the initial state (x0,y0), is respected.

This is the subtle difference between *Reset* and *Initialize.*



7.7 A final methodological remark

Some academics (see http://www.colos.org), which I respect, would complain that the model we have created for this example does not correspond to a real simulation. Their claim is that Nature doesn't solve differential equations and that we are not simulating the real phenomenon, *but a mathematical model of the phenomenon*.

Furthermore, they consider that there is an important pedagogical value in going a step closer in our simulation models to how Nature really does things.

If you think this claim is (at least, partially) right, take a look at the sample simulation *PredatorAndPreySimulation*. I tried myself to provide there a different approach to this phenomenon.

Appendices

Because the appendices pages can vary independently from the rest of the manual and, in order to facilitate future updates, the appendices are provided as a separate document.

This page intentionally left blank