Open Source Physics: [1] A User's Guide with Examples (Draft)

Wolfgang Christian

Includes

Physics Curricular Material by Mario Belloni

Tracker Video Analysis and OSP XML by Doug Brown

BQ Database by Anne Cox and William Junkin

Easy Java Simulations by Francisco Esquembre

September 6, 2006

[1]Open Source Physics is supported in part by National Science Foundation grants DUE-0126439 and DUE-0442481.

CHAPTER 17 Easy Java Simulations

The Open Source Physics project includes *Easy Java Simulations*, a high-level authoring tool that can be used both by expert programmers as a fast-prototyping utility, and by novices as a simple tool that will help them create their first simulations. This Chapter provides an overview of this application.

17.1 ■ INTRODUCTION

The OSP library provides a complete and articulated set of Java classes and utilities for programmers who want to create their own computational physics applications in Java. For these users, the library helps speed up the creation process.

However, OSP is just too powerful to reduce its potential use only to programmers. There are a great number of creative teachers (and students) who may not be fluent enough in Java to use the libraries as provided, but who could, if given the chance, develop effective and useful physics simulations.

For this reason, the OSP project aimed, from the very beginning, to find a way for non-programmers to access the concepts and to use the products of the OSP library. *Easy Java Simulations* is the answer to this goal.

Easy Java Simulations (*Ejs* for short) is a software authoring tool created in Java which sits on top of OSP libraries. It provides a simplified entry point for those who want to create Java applications or applets that simulate physical phenomena.

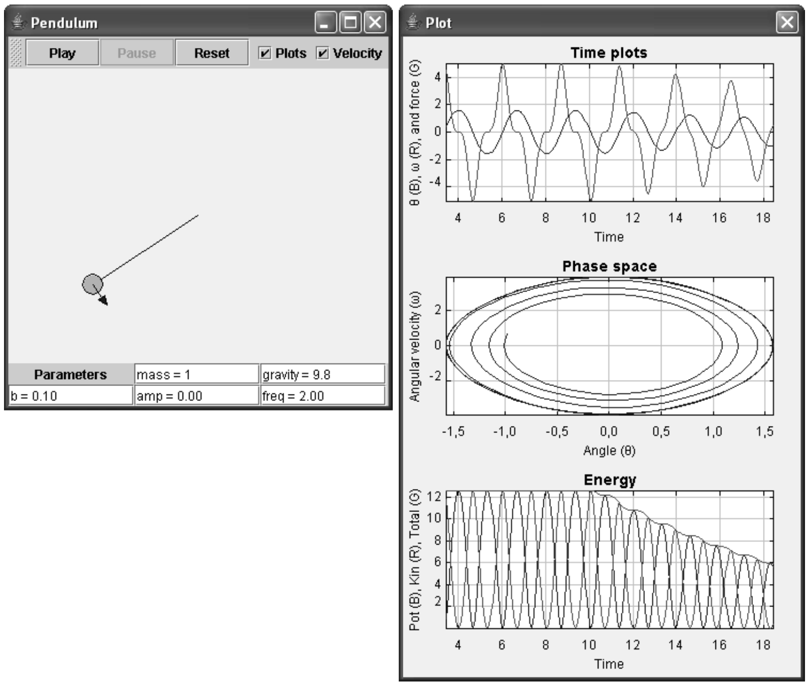
If you are an experienced Java programmer already or are on the way of becoming one, you may wonder why you need such an authoring tool when you can program directly? Even though it's true that direct programming gives you full control, before you decide to skip this Chapter completely, consider that you may want to learn more about a tool that can help you to:

- Quickly develop a prototype of an application in order to test an idea or algorithm.
- Boost the creation of sophisticated user interfaces with minimal effort.
- Create simulations whose structure and algorithms other people (especially non-programmers) can easily inspect and understand.
- Invite your students or colleagues (who may be new to Java) to create their own simulations.

- Automate production tasks such as preparing your simulations to be distributed using Web pages or Java Web Start technology.

This Chapter provides a general description of *Easy Java Simulations*and does not explain all of *Ejs*' features and possibilities, from the installation of the software to the Web distribution of the generated simulations. A detailed manual is, however, on the companion CD an on the Ejs web site. Instead, this Chapter is a short survey that describes how the tool works and how it simplifies the creation of professional OSP simulations.

We will illustrate how to use *Ejs* by working with one of the most famous Physics examples of all time, the simple pendulum. To get acquainted with *Ejs*' main features, we will begin by loading, inspecting, and running an existing basic simulation of this physical system. We will then extend the model to include damping and forcing and to improve the visualization of the phenomenon by including phase-space and energy graphs. Figure 17.1 shows the user interface of the simulation we will obtain once we have finished working with it.



Parameters	mass = 1	gravity = 9.8
b = 0.10	amp = 0.00	freq = 2.00

FIGURE 17.1 A simulation of a simple pendulum created with *Easy Java Simulations*. A damping term was introduced approximately at time $t = 10$ seconds.

17.2 ■ THE MODEL–VIEW–CONTROL PARADIGM MADE SIMPLER

We have mentioned the model–view–control (MVC) paradigm earlier in this guide (see Section 1.2). This was introduced as a way of structuring the different parts of a computer program and has proved to be successful in creating versatile applications in a clean, well-defined, and relatively simple, way.

Easy Java Simulations uses this same paradigm. It structures a simulation in two main parts: the model and the view. The model is the collection of variables that define the different possible states of the system under study, together with the algorithms that describe how these variables change in time or how they respond to user interaction. The view is the generic name that *Ejs* adopts for both the visualization of the phenomenon and the user interface of the application.

In essence, the *Ejs* view combines the control and view of the MVC paradigm. This simplifies things for beginners because in modern computer simulations the control is achieved through interaction of the user with the application's graphical user interface.

Because our simulations have mainly a pedagogical purpose, we will add to the model and view a textual (multimedia) part that is designed to contain a short introduction to the simulation or operating instructions for the user. Thus, an *Ejs* simulation consists of three main parts: the introduction, the model and the view. Figure 17.2 shows the interface of *Easy Java Simulations* (to which we have added some notes), which reflects this structure.

You are invited to run *Ejs* while you read this chapter. Although you can find detailed instructions on installing and running it in the manual included in the companion CD, here are some quick guidelines (for those who don't like reading manuals!). To install and run *Ejs*, follow the next steps:

Install Java 2 JDK. *Easy Java Simulations* is a Java program that compiles Java programs, hence it requires that you install Java Development Kit (Java 2 JDK) in your computer. The recommended version at the moment of this writing is 1.5.0_04.

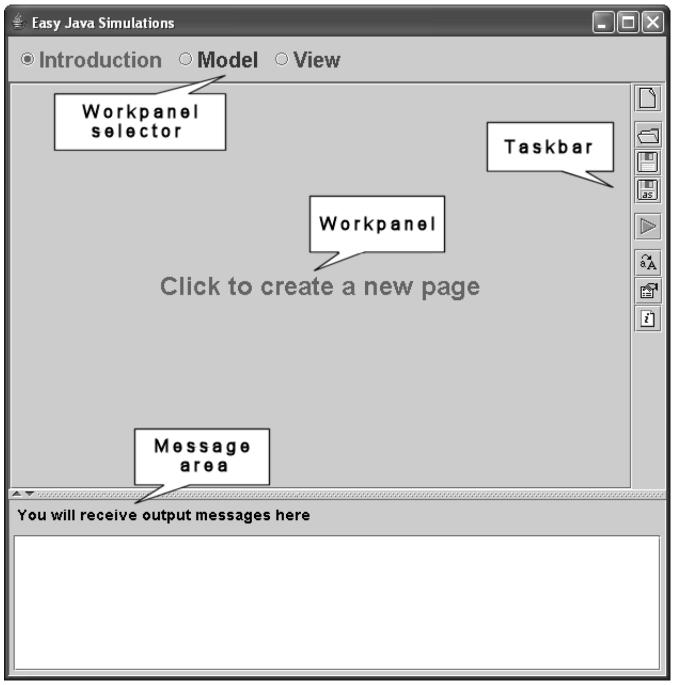
Copy *Ejs* to your hard disk. Installing *Ejs* requires only copying the **Ejs** directory to your hard-disk. This directory is usually distributed as a zip (compressed) file. Just uncompress this file to any suitable directory. (In Unix-like systems, the directory may be uncompressed as read-only. In this case, please enable write permissions for the whole **Ejs** directory.)

Run *Ejs*' console. *Easy Java Simulations* is now installed. The distribution includes an **EjsConsole.jar** file that runs a console that lets you do the final configuration for the installation and to run *Ejs* itself. Run this file by either double-clicking on it (if your system allows you to run it this way) or typing the command:

```
java -jar EjsConsole.jar
```

at a system terminal window. You should get the window shown in Figure 17.3.

Tell *Ejs* where you installed the JDK. (This is only required in Windows



You will receive output messages here

FIGURE 17.2 *Easy Java Simulations* user interface (with annotations).

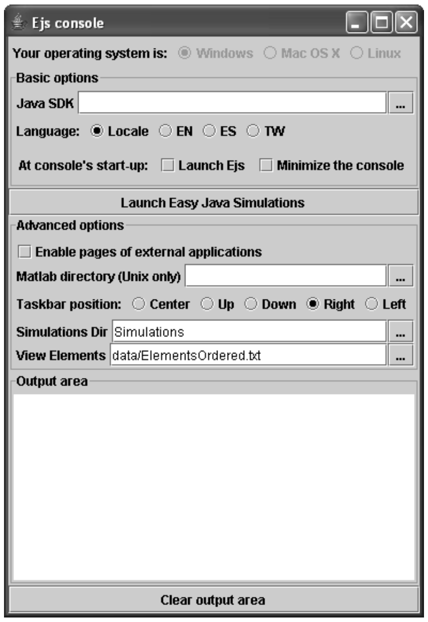
and Linux.) Because you may have installed Java 2 JDK in any directory, you will need to enter this directory in the `Java JDK` field of the console.

Run *Ejs*. Just click on the `Launch Easy Java Simulations` button of the console.

That's it! You should get the interface displayed in Figure 17.2. Again, you will find more detailed instructions in the CD and also in *Ejs*' home page `http://fem.um.es/Ejs`. [1]

As you can see in Figure 17.2, the interface of *Ejs* is rather basic. This was a deliberate design decision. We wanted to make clear from the very beginning that *Easy Java Simulations* is simple. Hence, we avoided providing a large and overwhelming number of icons and menu entries in the *Ejs* interface. Despite its simplicity, however, the tool has everything that it needs to build controls, models, and views.

[1]For operating system purists, *Ejs*' installation includes three script files that launch *Ejs* in the different platforms: **Ejs.bat** for Windows, **Ejs.macosx** for Mac OS X, and **Ejs.linux** for Linux. Before running the file corresponding to your operating system, you may need to slightly edit the script to correctly set the `JAVAROOT` variable (which is defined in the first lines of the script) that points to the directory where you installed the Java JDK.


Ejs console

Your operating system is: (● Windows) ○ Mac OS X ○ Linux

Basic options

Java SDK

Language: (● Locale) ○ EN ○ ES ○ TW

At console's start-up: ☐ Launch Ejs ☐ Minimize the console

Launch Easy Java Simulations

Advanced options

☐ Enable pages of external applications

Matlab directory (Unix only)

Taskbar position: ○ Center ○ Up ○ Down (● Right) ○ Left

Simulations Dir: Simulations

View Elements: data/ElementsOrdered.txt

Output area

Clear output area

FIGURE 17.3 The *Ejs*' console that will help you run *Easy Java Simulations*.

We start exploring the interface by looking at the set of icons on the right hand side taskbar of Figure 17.2. The taskbar provides an icon for each of the main functionalities of *Ejs*. We will explain the purpose of some of these icons in this chapter, but their meaning and use should be rather natural.

You will also notice in Figure 17.2 that there is a blank area at the lower part of the window with a header that reads "You will receive output messages here". This is a message area that *Ejs* uses to display information about the results of the actions we ask it to take.

Finally, the most important part of the interface is the central area of the interface (labeled the `Workpanel` in the figure), and the three radio buttons on top on it, `Introduction`, `Model` and `View`. These radio buttons are used to display the introduction, the model and the view in the workpanel.

17.3 ■ INSPECTING AN EXISTING SIMULATION

We can better understand the role of the different parts of a simulation, and the utility of the different panels of *Ejs*, by loading and inspecting an existing simulation. We choose for this a basic implementation of the simple pendulum that we created for your perusal and that is included in the distribution of *Ejs* in the companion CD.

If you click on the `Open` icon of the taskbar, 🗀, a file dialog box will open from which you can load a simulation file. See Figure 17.4.

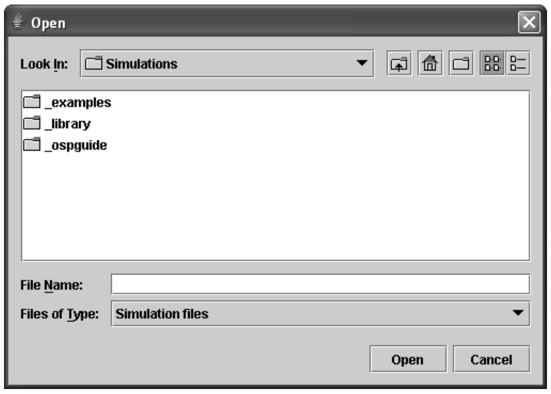


FIGURE 17.4 File dialog box used to load an existing simulation file.

The file dialog box first displays the contents of *Ejs*' working directory called **Simulations**. From there you can access any directory or file in the standard way. Open the directory called **_ospguide** and select from it the file called **PendulumBasic.xml**. Click the `Open` button and *Ejs* will load a basic version of the simulation of a simple pendulum.

The interface of *Ejs* will change noticeably. It will load the different parts of the simulation file in the corresponding panels, allowing us to see how the simulation has been designed. It will change its title to include the name of the simulation file, and, finally, it will also display a message, in the message area, reporting that the file has been loaded.

Two new windows will also appear. You can see them in Figure 17.5. They correspond to the view of the simulation, which we will describe a bit later. [2] For the moment, however, we will concentrate on the interface of *Ejs* itself.

What we see in this interface depends on which part of the simulation was visible when we loaded the file. Because we are interested in learning how to use *Ejs* to specify a real simulation, we will inspect the different parts of the simulation in turn by browsing the different panels of *Ejs*.

The Introduction

The first panel is the introduction of the simulation, shown in Figure 17.6, which consists of a html page with a short introduction to the problem. This part of the interface of *Ejs* provides a simple editor that can be used to both visualize and edit standard html pages. Right now, we see the editor in its read-only mode. We will learn in Section 17.5 how to set it to edit mode and actually change its contents.

[2] To be more precise, these two windows are, as their titles state, `Ejs windows`. That is, windows that *Ejs* displays to help the author configure the view. Hence, they are really mock-ups of what the real

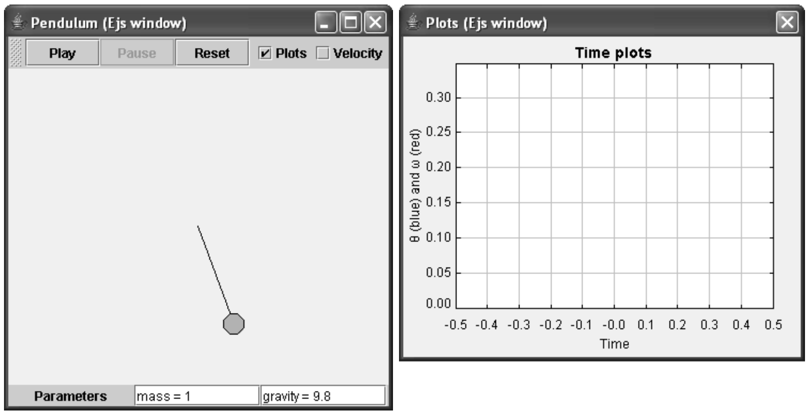
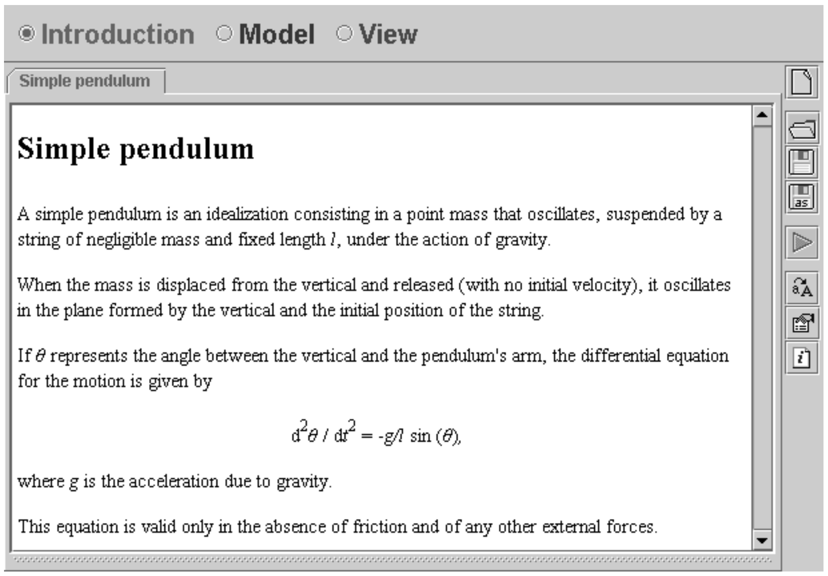


FIGURE 17.5 The two windows for the view of the simulation.



Simple pendulum

A simple pendulum is an idealization consisting in a point mass that oscillates, suspended by a string of negligible mass and fixed length l, under the action of gravity.

When the mass is displaced from the vertical and released (with no initial velocity), it oscillates in the plane formed by the vertical and the initial position of the string.

If θ represents the angle between the vertical and the pendulum's arm, the differential equation for the motion is given by

$$d^2\theta \,/\, dt^2 = -g/l \,\sin(\theta),$$

where g is the acceleration due to gravity.

This equation is valid only in the absence of friction and of any other external forces.

FIGURE 17.6 The introduction panel for the simulation of a simple pendulum.

The Model

The second part of the simulation, the model, is more interesting. As you may expect, this is the part where all the physics goes. The physical model of a simple pendulum without friction and without an external driving force is contained in

view will look like, but they are not operative. Notice also that this view is different from the view of the final simulation we will obtain at the end of this chapter, the one displayed in Figure 17.1.

the differential equation:

$$\frac{d^2\theta}{dt^2} = -\frac{g}{l}\sin(\theta), \tag{17.1}$$

where the variable θ corresponds to the angle between the pendulum's arm from the vertical, g is the acceleration due to gravity, and l is the length of the pendulum.

This second-order, non-linear, ordinary differential equation (ODE) must be solved numerically because there is simply no way to express its solution in terms of elementary functions. [3] To this end, we need to express the second-order differential equation as an equivalent system of two first-order ODEs, introducing the auxiliary variable ω, the angular velocity:

$$\frac{d\theta}{dt} = \omega \tag{17.2}$$

$$\frac{d\omega}{dt} = -\frac{g}{l}\sin(\theta). \tag{17.3}$$

Solving this system of ODEs for the given parameters will give us the evolution of the angular position (and velocity) of the system in time.

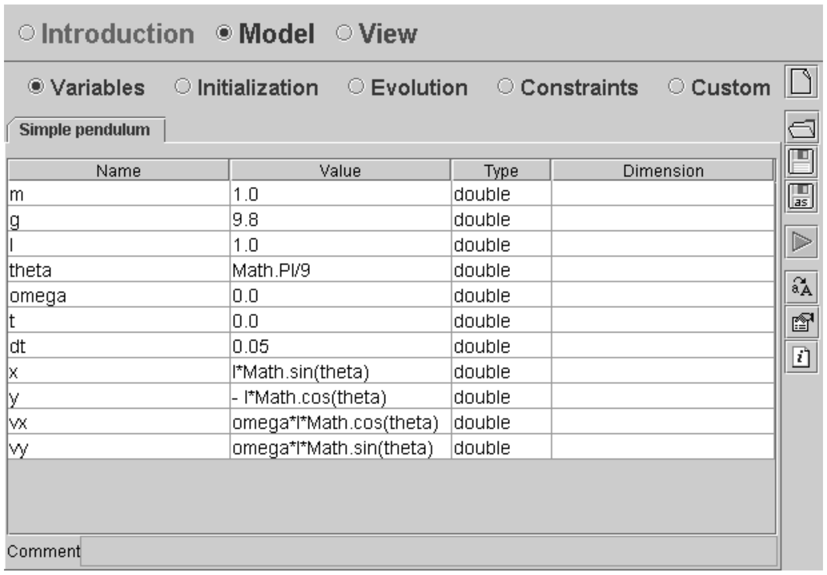
The second part of *Easy Java Simulations*, the model, offers us a set of five subpanels that can be used to accomplish the tasks needed to solve this problem. These five subpanels are: `Variables`, `Initialization`, `Evolution`, `Constraints`, and `Custom`. In the *Ejs* pendulum example, select the `Model` radio button in order to view the model panel. We now consider the five subpanels one after the other.

Declaring the Variables

In the context of a computer program or simulation, the state of a physical system is determined by the values of a set of variables. Accordingly, to begin modeling our simple pendulum, we must provide *Ejs* with the set of variables that defines the state of the physical system. This is done by editing a simple table of variables in the `Variables` subpanel of the model part of the interface of *Ejs*. See Figure 17.7.

In this table we have declared all the variables involved in the differential equations as well as other parameters, such as the mass, `m`, (which we don't need for the moment, but that we will need later on) and the time interval, `dt`, at which we want to obtain information from the system. Finally, you will notice that we have also declared the variables `x`, `y`, `vx`, and `vy`, that hold the position and the velocity of the pendulum's bob. They will serve to configure the view to display a realistic visualization of the pendulum.

[3] To be more precise, the case we consider here in the absence of friction and of any other forces can be explicitly solved using elliptic functions. However, we adopt the numerical approach right from the start in order to be able to deal later with the more general case.


○ Introduction ◉ Model ○ View
◉ Variables ○ Initialization ○ Evolution ○ Constraints ○ Custom
Simple pendulum
Comment

Name	Value	Type	Dimension
m	1.0	double	
g	9.8	double	
l	1.0	double	
theta	Math.PI/9	double	
omega	0.0	double	
t	0.0	double	
dt	0.05	double	
x	l*Math.sin(theta)	double	
y	- l*Math.cos(theta)	double	
vx	omega*l*Math.cos(theta)	double	
vy	omega*l*Math.sin(theta)	double	

FIGURE 17.7 Table of variables that describe the state of a simple pendulum.

If you want information about the role of a particular variable, you can select it in the table and the comment field at the bottom of the page will display a short description of that variable.

Initializing the Model

The system must be initialized to a valid state before letting the time run. There are two basic ways of initializing variables in *Ejs*. The first one is using the corresponding cells of the column `Value` in the table of variables. Just type a constant value, or a simple expression, and it will be assigned to the variable at start-up. This is the option we have used for this simulation.

A second possibility, which is required if your program needs to do some more complex computations to initialize the system, is to use the `Initialization` subpanel provided by *Ejs*. In this subpanel you can type the Java code for the algorithm that will compute the correct values for the variables that require these extra computations. *Ejs* helps to keep this process simple, because you just need to write the Java sentences that contain the algorithm for your computations, and *Ejs* will automatically wrap these algorithms into a Java method and will take care of calling it at start-up or whenever you reset the simulation.

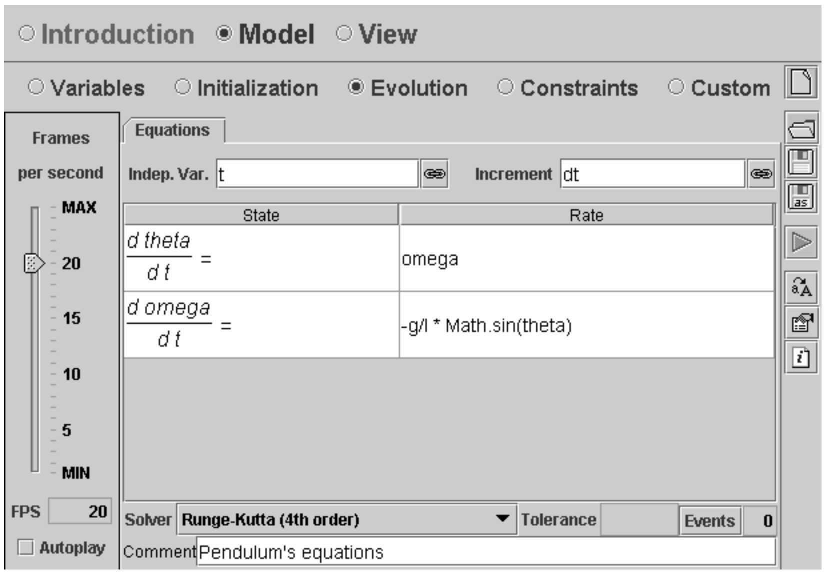
Because the system has been completely initialized in the table of variables, the initialization panel remains empty. We'll have the opportunity to show how to write such a page of Java code when we cover constraints.

The Evolution of the Model

Specifying the evolution of the model consists of providing the algorithms that describe how the state of the system changes in time. That is, what happens to the values of the variables of the system when it evolves in time. In our case, this corresponds to solving numerically the differential equations of the model.

Ejs offers two possibilities for this. The first one is a plain editor for Java code where the author can write directly the numerical algorithm required. You would typically use this option if your main interest is simply numerical algorithms. However, because it is a common situation that the evolution of a system is specified by a system of ODEs, and writing the code to solve this type of problem with accuracy can be tedious (if not difficult), *Ejs* also offers a dedicated built-in editor. This editor allows us to easily enter the system of ODEs, and it automatically generates the Java code (based on OSP numeric classes) corresponding to some of the most popular numerical algorithms to solve the equations.

This second possibility is the one we chose for our system. If you inspect the `Evolution` subpanel of *Ejs*, you will see that it contains our system of ODEs in this specialized editor, as shown in Figure 17.8.



Indep. Var. t Increment dt

State	Rate
$\frac{d\,theta}{d\,t}$ =	omega
$\frac{d\,omega}{d\,t}$ =	-g/l * Math.sin(theta)

FPS 20

Solver Runge-Kutta (4th order) Tolerance Events 0

Comment Pendulum's equations

FIGURE 17.8 Equation editor for our simulation with the pendulum's system of ODEs.

One advantage of this editor is that it displays the system of ODEs in a very familiar way, one that your students can easily recognize and that is easy to understand and modify. The second advantage is that the editor takes care of the worst part of solving the equation: coding the algorithm. Solving differential equations numerically is a sophisticated task, but *Ejs* (with the help of OSP classes) has automated it in a very convenient way.

You will notice in Figure 17.8 that we have chosen `t` to be the independent variable in the model and `dt` to be the increment for it in each evolution step. This means that the evolution page should solve the equation to move from the current instant of time at `t`, to the next instant of time at `t+dt`.

We have chosen the standard 4th-order Runge-Kutta algorithm, as the `Solver` field immediately below the equations indicates. You will also notice that there are two extra fields, one called `Tolerance` (that is used only for adaptive algorithms), and one called `Events`. This last field is used to define and handle state-events, such as collisions, that may occur in the life of the differential equation. (Events are described in detail in the manual on the CD.) In our case, the simple pendulum model includes no events.

All together, this page describes, using the editor for ODEs, what will happen as time passes: the differential equations will be solved numerically for an increment of `dt` of the independent variable. The controls you see in the left-hand side of Figure 17.8 are used to tell *Ejs* how often the evolution should take place when the simulation runs. As the figure shows, we have instructed *Ejs* to run the evolution 20 times, or frames, per second. This, together with the value of 0.05 that we used for `dt`, results in a simulation which runs (approximately) in real time.

Finally, there is a checkbox labeled `Autoplay`. This instructs *Ejs* to run the evolution as soon as the program runs. We left it unchecked because we want to offer the user the possibility of changing the position of the pendulum and then to click on a button that will start the animation.

Constraints Among Variables

The fourth subpanel of the model is called `Constraints`. This panel is used to write Java code that establishes fixed relationships among variables. Consider, again, the pendulum example as shown in Figure 17.9.

The evolution of our model solves the ODEs in terms of the angular magnitudes, `theta` and `omega`. However, we are interested in displaying on the screen the actual position of the pendulum and its velocity vector, and, for this, we need the corresponding cartesian coordinates of both position and velocity. These can be easily derived from the angular magnitudes using the formulas:

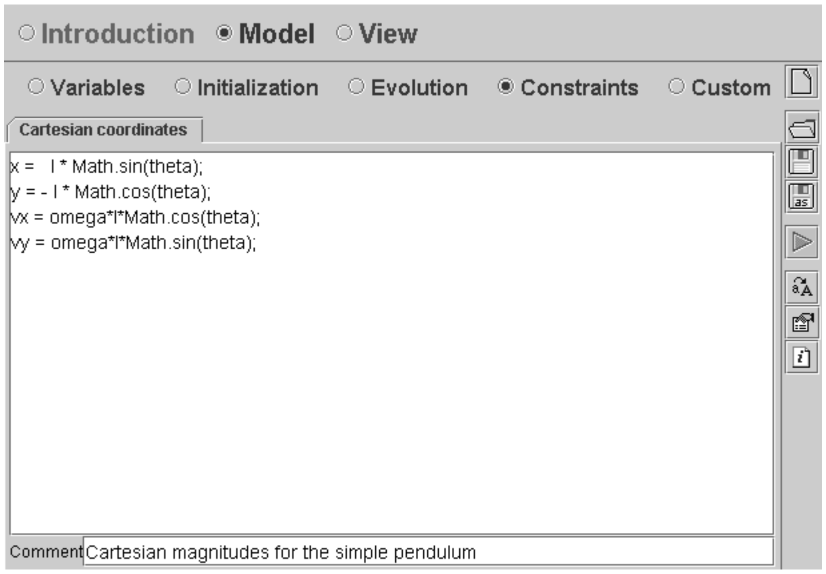
$$x = l\ \sin(\theta) \tag{17.4}$$

$$y = -l\ \cos(\theta) \tag{17.5}$$

$$v_x = \omega\, l\ \cos(\theta) \tag{17.6}$$

$$v_y = \omega\, l\ \sin(\theta). \tag{17.7}$$

This is what we call "fixed relationships among variables." This means that, once we know the values of l, θ, and ω, the other variables can be easily obtained using the expressions above. Our constraints consist of translating these expressions into Java so that *Ejs* can use them whenever it is needed.



```
x =   l * Math.sin(theta);
y = - l * Math.cos(theta);
vx = omega*l*Math.cos(theta);
vy = omega*l*Math.sin(theta);
```

Cartesian magnitudes for the simple pendulum

FIGURE 17.9 Constraints pages that compute the cartesian magnitudes of a simple pendulum.

Notice that the page contains only the Java code for our expressions. *Ejs* will take care of wrapping this code into a Java method and automatically calling this method whenever it is necessary. This simplifies our programming task.

Now comes a subtle point. One of the questions most frequently asked by new users of *Ejs*: “Why don’t we write these equations into the evolution instead of in the constraints?”. The reason is that this relationship among variables must *always* hold, even if the evolution is not running. It could very well happen that the simulation is paused and the user interacts with the simulation to change the angle `theta` (or `omega`, or `l`). If we write the code to compute the cartesian magnitudes in the evolution, the values for the variables `x`, `y`, `vx`, and `vy` will not be properly updated, because the evolution is only evaluated when the simulation is playing.

Constraint pages, on the other hand, are always automatically executed after the initialization (at the beginning of the simulation), after every step of the evolution (when the simulation is playing), and each time the user interacts with the simulation. Therefore, any relationship among variables that we code in here will always be verified.

> You could argue here that, since constraints are evaluated also at start-up, there was no reason to initialize the variables `x`, `y`, `vx`, and `vy` in the table of variables since the evaluation of the constraints will assign them the correct values at start-up. You would be almost right. There is still a reason for doing what we did, though.
> *Ejs* doesn’t evaluate constraints itself (the generated simulation does), but it evaluates expressions in the `Value` column of the table of variables. Hence, we wrote the initial expressions for these variables so that the pendulum’s

bob appears at the right place in the mock-up of the view displayed. But it is true that, in general, constraints can also be used to initialize variables.

Custom Pages of Code

The final subpanel of the model is called `Custom`. This panel can be used by the author of the simulation to define his or her own Java methods. Different from the rest of panels of the model, which play a well-defined role in the structure of the simulation, methods created in this panel must be explicitly used by the author in any of the other parts of the simulation. Again, in our example, this panel is empty and is not displayed.

Although *Ejs* is designed to make programming as simple as possible and includes the typical tools an author may need, it also opens a way for programmers to use their own Java libraries. The custom panel of the model offers a simple mechanism to add external Java archives of compiled classes, packed in jar or zip form, to your simulation. The procedure is explained in detail in the manual.

The Model as a Whole

Our description of the model is ready, and we can look at it as one unit to describe the integral behavior of all the subpanels of the model.

To start the simulation, *Ejs* declares the variables and initializes them, using both the initial values specified in the table of variables and whatever code the user may have written in the initialization subpanel. At this moment, *Ejs* also executes whatever code the user may have written in the constraint pages. This is so that all dependencies between variables are properly evaluated. The system is now correctly initialized.

When the simulation plays, *Ejs* executes the code provided by the evolution subpanel and, immediately after, the possible constraints (for the same reason as above). Once this is done, the system will be ready for a new step of the evolution, which it will repeat at the prescribed speed (number of frames per second).

As mentioned already, note that methods in the custom subpanel are not automatically included in this process.

This simple mechanism provides a basic, but very effective, structure for novices (and experts!) to build their simulations. The author just fills in the subpanels of the model, usually from left to right, and *Ejs* handles the pieces, automatically taking care of all technical issues required (such as multitasking and synchronization).

The View

We now turn our attention to the view of the simulation. Recall that two new windows appeared when we loaded the simulation (see Figure 17.5). To learn how this view has been constructed, we can inspect the `View` panel of *Ejs*. Figure 17.10 displays this panel, where two frames, each with several icons, are shown. The frame on the right-hand side displays the set of graphical elements that *Ejs* offers to authors for the creation of a view, grouped by functionality. The frame on the left-hand side shows the actual elements that have been used for this particular simulation.

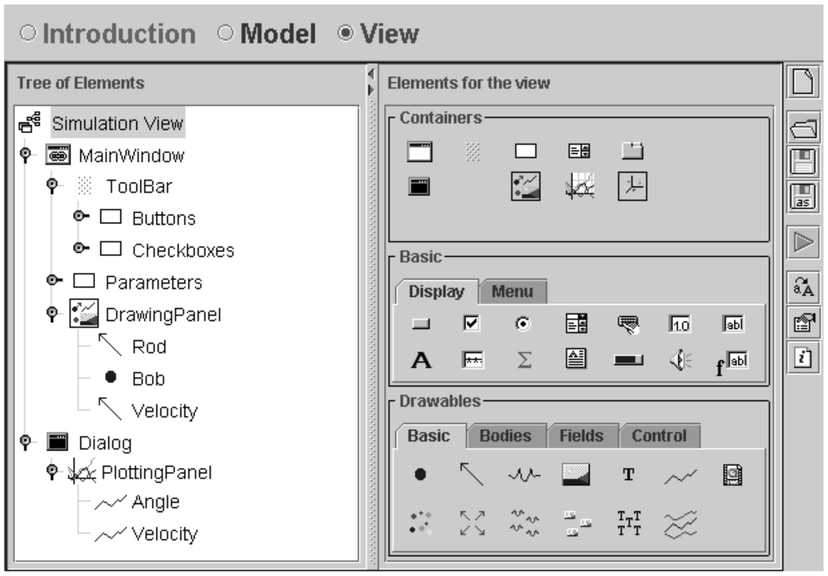


FIGURE 17.10 Tree of elements for the simulation (left) and set of graphical elements (right) of *Ejs*.

The view panel of *Ejs* can be considered as an advanced drawing tool which specializes in the visualization of scientific phenomena and its data and user interaction. Obviously, to completely master the creation of views, an author needs to become familiar with all the graphical elements offered and what they can do. (A description of all possible view elements is out of the scope of this chapter, but a complete reference can be found on the companion CD.)

Creating a view with *Easy Java Simulations* takes two steps. The first step is to build a tree-like diagram of the objects (elements) that will make up the user interface. Each element is designed for a given graphical or interactivity task, and our job is to select the elements that we need and combine them appropriately to build our view. Selecting and adding new elements to a view is done in a simple "click-and-use" way which we will illustrate in Section 17.5.

Some elements are of a special family called *containers*, which can be used to group other elements, thus forming the tree-like structure of the elements shown in the figure.

The second step, less evident but also simple, consists of customizing the selected view elements by editing their so-called *properties*. Properties are internal fields of an element that can be changed to make the element look and behave in a particular way. The key point is that properties can be given constant values (for instance to customize fonts and colors), but they can also be *linked* to variables of the model (typically for positions, sizes and labels with numerical displays).

Because linking is a two-way mechanism, this second possibility is what really turns the view into a dynamic, interactive visualization of the physical phenomenon. Hence, once an element property is linked to a model variable, any change in the variable (due for instance to the evolution of the model) is automatically reported to the view element which changes its graphical aspect accordingly. But, also, if the user interacts with any view element to modify any of its properties (typically doing a gesture with the mouse or keyboard), the change is automatically reported back to the model variable, therefore changing the state of the system.

This basic mechanism is a very simple and effective way to design interactive user interfaces. Let us see an example of how it works in practice. Select the panel for the view and right-click on the element called `Bob` of the tree of elements of our simulation. This element corresponds to the circle displayed as the bob of the pendulum. The popup menu shown in Figure 17.11 will appear.

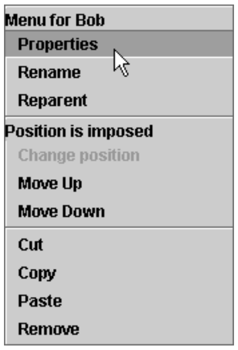


FIGURE 17.11 Popup menu for the element `Bob` of the view.

Select the option `Properties` from this menu (the one highlighted in the figure) and a new window will appear with the table of properties for this element. [4] All that is required is to edit this table according to our needs. Figure 17.12 reflects the properties we defined for this particular element.

This table of properties illustrates very well all the possibilities offered by element properties. In the first place, you can see in the figure that some properties are given constant values. For instance, those which specify the color, drawing style, and size of the element (which will be displayed as a cyan-colored ellipse of size (0.2,0.2) units). You can also see that the element is enabled, that is, that it will respond to user interaction.

Secondly, you will observe that other properties of the element are given the value of model variables. In particular, the properties `X` and `Y`, which correspond to

[4]Double-clicking on the element's node in the tree is a shortcut for this option.

the center of the circle in its parent drawing panel, have been linked to the model variables `x` and `y` by the simple fact of typing their names in the corresponding property field. This connection is the magic. Automatically, the circle will move according to the successive values of the model variables `x` and `y` when the simulation runs. Also, if the user drags the element Bob with the mouse, the model variables `x` and `y` will be accordingly changed.

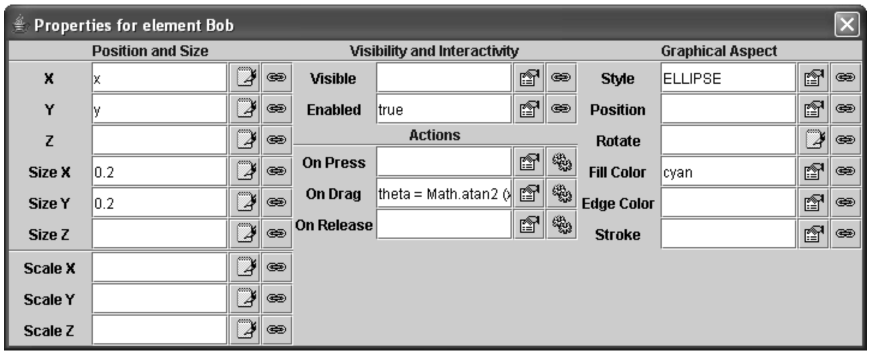
As described above, every time the user interacts with the simulation's view, *Ejs* will also automatically execute the constraints of the model. This ensures that any change that the interaction caused to variables linked to properties is correctly propagated to other variables which may depend on those.

Finally, there is a third type of property that needs to be taken care of for this example to work properly. Recall that the model computes primarily the values of the angular magnitudes, and that we wrote constraints to make sure that the cartesian magnitudes (which includes `x` and `y`) were readily computed to match those. This means that if we interact with the pendulum bob to change the values of `x` and `y`, the change will be overwritten by the constraints unless the change does affect the angular variables. This can be easily taken care of by means of the (somewhat special) *action* properties of the element. These correspond to pieces of Java code that the computer will evaluate whenever the prescribed interaction takes place.

In our case, we need to edit the action property called `On Drag`, which is evaluated every time we drag the element on the screen. We have set this property to execute the following sequence of sentences: [5]

```
theta = Math.atan2 (x,-y);
l = Math.sqrt (x*x + y*y);
omega = 0.0;
```

[5]Unlike other properties, action properties can span more than one line. When this happens, the property field changes its background color slightly. To better display this code, we can click on the first button to its right, 📝, and an editor window will display it more clearly.



Properties for element Bob

Position and Size		Visibility and Interactivity		Graphical Aspect	
X	x	Visible		Style	ELLIPSE
Y	y	Enabled	true	Position	
Z		**Actions**		Rotate	
Size X	0.2	On Press		Fill Color	cyan
Size Y	0.2	On Drag	theta = Math.atan2 (x	Edge Color	
Size Z		On Release		Stroke	
Scale X					
Scale Y					
Scale Z					

FIGURE 17.12 Table of properties for the element `Bob` of the view.

The first two statements use the (new) values of the `x` and `y` variables to compute the new length of the pendulum and its angular position. We have added a third sentence that sets the angular velocity to zero, because we want the motion to re-start from rest. All together, these sentences help set up a new state for the model of the system.

The three types of customization we did to the properties of this view element illustrate how *Easy Java Simulations* combines the creation of the control and the visualization tasks of a simulation in one single process (the view). By using either constants, model variables, and Java expressions and sentences, we can configure a user interface that is used simultaneously to display data, to provide input, and to execute control actions on the simulation.

You can inspect the properties of other elements of this view to become familiar with the different types of view elements offered by *Ejs* and their properties. Of particular interest are the elements called `Angle` and `Velocity`, which correspond to time plots of these magnitudes. For each element, you can click on 🗒, the first button to the right of each property box, to bring up an editor for that property. For example, the `Style` property editor will allow you to select the shape of the element, and the `Fill Color` property editor will allow you to select the color from a palette.

17.4 ■ RUNNING A SIMULATION

Once we have inspected the different parts of the simulation, we are ready to run it. Here goes a warning that may arrive too late if you are running *Ejs* while reading these pages: don't try to run the simulation by clicking on the buttons of the windows in Figure 17.5! Recall that these were just "Ejs windows". Hence, they are not part of the real simulation, but just a mock-up of what the real view will look like. Their purpose is to help the author design the view, but they are not operative.

To actually run the simulation, you need to click on the `Run` icon of *Ejs*' toolbar, ▶. When you do this, *Easy Java Simulations* collects all the information provided in its panels and subpanels, and constructs a complete, independent simulation out of it, taking care of all the required technical subtleties. This includes generating the complete Java source code for the simulation, compiling it, and packing it into a single jar file. Finally, it also runs this file, which will initialize the model and display the simulation's view in the computer screen.

This view is now fully interactive. You can drag the pendulum to any desired position, change any of its parameters, and then click the `Play` button. The pendulum will oscillate as the model solves the underlying differential equations. The view will display both the pendulum's oscillations and the time plot of the position and (optionally) the velocity. See Figure 17.13.

Simulations created with *Easy Java Simulations* are independent of it once generated. This means that final users don't need to install *Ejs* to run the simulations. They simply double-click the jar file. Moreover, when you have success-

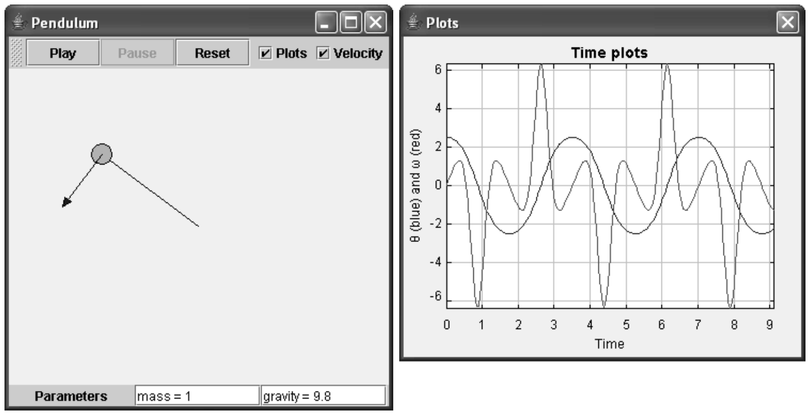


FIGURE 17.13 The simple pendulum displaying oscillations with a big amplitude. The largest plot corresponds to the angular velocity.

fully run a simulation once, you'll find everything you need to run and distribute the simulation in the **Simulations** directory. We now briefly discuss the different ways to run a simulation created with *Easy Java Simulations*, together with some remarks about its distribution.

Running the Simulation as an Application

After running the simulation, you will find in the **Simulations** directory a jar file with the same name as the simulation file we first loaded (if only with its first letter in lowercase, a style custom in Java programming). Recall that the name of this file was **PendulumBasic.xml**. Hence, you will find there a jar file called **pendulumBasic.jar**.

This is a self-running jar file. Hence, if your operating system is properly configured, you will be able to run the simulation by double-clicking on the jar file icon. (Windows and Mac OS X are usually automatically configured for this if the Java runtime environment is installed.) If double-clicking won't work, then you can still use the launch file called **PendulumBasic.bat**. This has been generated by *Ejs* specifically for your operating system and can be executed like any other batch (Windows) or shell script (Unix-like systems) on your operating system.

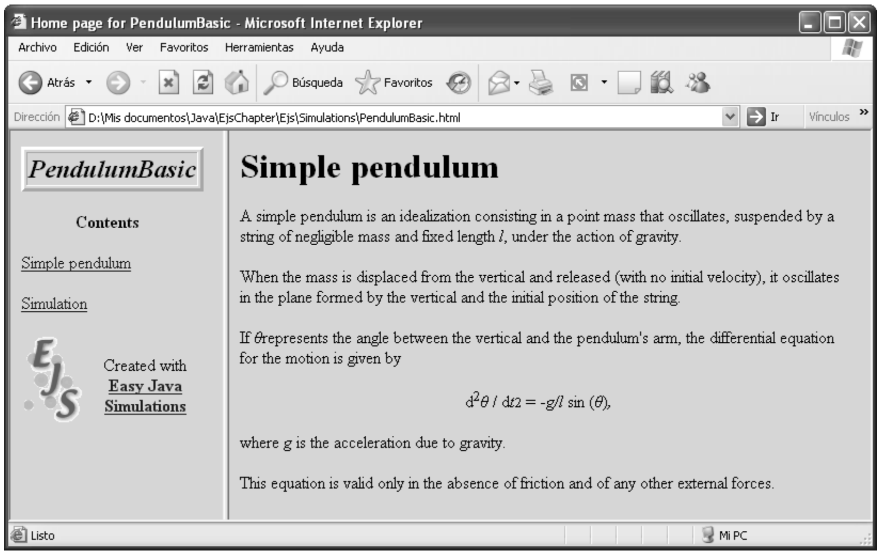
This jar and launch file will work correctly assuming that you have the Java runtime environment (JRE) installed in your system and that you run them from within the **Simulations** directory. If you want to move your simulation to a different directory or computer, read the distribution notes below.

Running the Simulation as an Applet

A second possibility to use the simulations created with *Ejs* is to run them as Java applets from within html pages. This is a very attractive possibility because it opens the world for distributing simulations over the Web (see also the note about the Java Web Start technology below).

This possibility is also taken care of by *Easy Java Simulations*. Every time you run a simulation from *Ejs*, it also generates the html pages required to wrap the simulation in form of an applet. More precisely, it will create a complete set of html pages, one for each of the introduction pages (those in the introduction panel of *Ejs*) and one that contains the simulation as an applet. It finally creates a master html file that structures all the others using a simple set of frames.

All these html files are easily identified because they begin with the same name as your original simulation file. In particular, the name of the master file is the same as that of the simulation file. In our case, **PendulumBasic.html**. If you load this file in a Java-enabled browser, you will see something like Figure 17.14.


Home page for PendulumBasic - Microsoft Internet Explorer
Archivo Edición Ver Favoritos Herramientas Ayuda
Atrás
Búsqueda
Favoritos
Dirección
D:\Mis documentos\Java\EjsChapter\Ejs\Simulations\PendulumBasic.html
Ir
Vínculos
PendulumBasic
Contents
Simple pendulum
Simulation
Created with
Easy Java
Simulations
Listo
Mi PC

Simple pendulum

A simple pendulum is an idealization consisting in a point mass that oscillates, suspended by a string of negligible mass and fixed length *l*, under the action of gravity.

When the mass is displaced from the vertical and released (with no initial velocity), it oscillates in the plane formed by the vertical and the initial position of the string.

If θrepresents the angle between the vertical and the pendulum's arm, the differential equation for the motion is given by

$$d^2\theta \;/\; dt2 = -g/l \sin(\theta),$$

where *g* is the acceleration due to gravity.

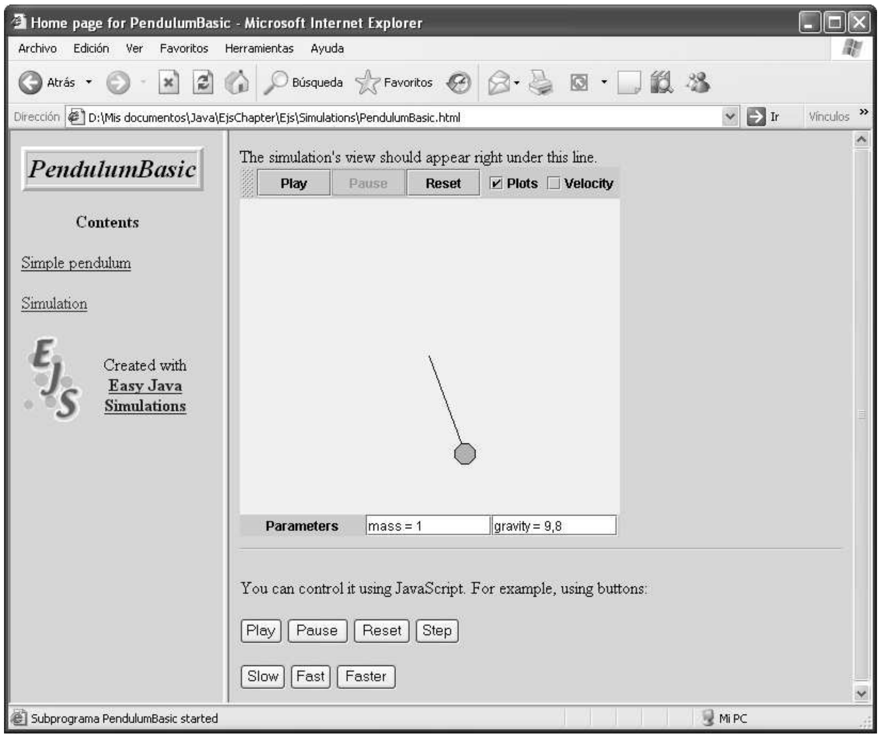
This equation is valid only in the absence of friction and of any other external forces.

FIGURE 17.14 The set of html pages created for our simulation.

Notice that the frame on the left displays a table of contents that includes the introduction page that we created for our simulation (which is also shown in the right frame) and a second link for the simulation itself. If we click on this link, the frame to the right will change to display a html page with the simulation embedded as an applet. See Figure 17.15. (The dialog window with the time plots is displayed separately. We don't reproduce it here.)

Notice, finally, that the html page that includes the simulation also displays a set of buttons that can be used to control the simulation using JavaScript. [6]

[6]JavaScript is a scripting language that can be used in html pages for simple programming tasks. The use of JavaScript to access methods of simulations created with *Ejs* is described in the *Ejs* manual.


Home page for PendulumBasic - Microsoft Internet Explorer
Archivo Edición Ver Favoritos Herramientas Ayuda
Atrás Búsqueda Favoritos
Dirección D:\Mis documentos\Java\EjsChapter\Ejs\Simulations\PendulumBasic.html
Ir Vínculos
PendulumBasic
Contents
Simple pendulum
Simulation
Created with Easy Java Simulations
Play Pause Reset Plots Velocity
Parameters mass = 1 gravity = 9,8
Play Pause Reset Step
Slow Fast Faster
Subprograma PendulumBasic started
Mi PC

The simulation's view should appear right under this line.

You can control it using JavaScript. For example, using buttons:

FIGURE 17.15 The simulation running as an applet.

Simulations created with *Ejs* require Java 2 to run. Thus, your browser will need to have a recent plug-in installed to display them properly. We recommend the Java plug-in version 1.5.0_04.

Distributing a Simulation

As we said before, simulations created with *Easy Java Simulations* are independent of it once generated. However, the simulation will need to use a set of library files that include all the compiled OSP classes that provide the background functionality for your simulation, from numerical methods to the visualization elements. This library can be freely distributed and is contained in the **_library** subdirectory of the **Simulations** directory. Finally, if you designed your simulation to use any additional file, such as a GIF image or sound file, you will need to distribute this too along with the simulation.

With this said, the distribution process is quite simple. You just need to copy the required files and the **_library** directory to the distribution media or Web server. Simulations created with *Ejs* can be distributed using any of the following ways:

Web server Just copy the jar, html, and auxiliary files for the simulation in a suitable directory of your Web server. Finally, copy the **_library** directory into the same directory as your simulation on the Web server. Recall that your users will need to have a Java 2 plug-in enabled web browser to properly display the simulation.

CD-ROM This procedure is similar to the previous one. Just copy the jar, html, and auxiliary files, and the **_library** directory into your distribution media. Your users will need to have the JRE installed to run your simulation.

Easy Java Simulations is compatible with **Launcher** (see Chapter 15) and the *Ejs*' installation includes a copy of **LaunchBuilder** that will scan your **Simulations** directory and will automatically generate the XML file that you need to run your simulations using **Launcher**. You can then distribute this XML file along with your simulation files, so that your users can use **Launcher** to run them.

Java Web Start This is a technology created by Sun to help deliver Java applications from a Web server. The programs are downloaded from a server at a single mouse click, and they install automatically and run as independent applications. *Easy Java Simulations* is prepared to help you deliver your simulations using this technology, and it can automatically generate a Java Web Start jnlp file for your simulation. Details are provided in the manual. Your users will need to have the JRE installed, which includes Java Web Start. Finally, your server will need to report a MIME type of `application/x-java-jnlp-file` for any file with the extension **jnlp**.

Together with *Ejs* A final possibility that is worth considering is that of distributing your simulation files together with *Ejs*. That is, asking your users to run the simulations using *Ejs* itself. This requires your users to learn how to use *Ejs*, if only basically to load and run the simulations. But it has, in our opinion, the enormous advantage that your users can not only run the final simulation, but also inspect it in detail and learn how you actually simulated a given phenomenon. This possibility also offers a simple way for students to begin programming to simulate physical phenomena, which we find of great pedagogical value.

17.5 ■ MODIFYING A SIMULATION

If you read up to this point, you should already have a general impression on how *Easy Java Simulations* works, how it can be used to create a simulation, and how to run and distribute the simulations you create with it. In this section we will modify the simulation of the simple pendulum to include new features. This will further illustrate the operating procedures required to work with the different panels of *Ejs*.

In particular,

1. We will modify the model to add friction and an external driving force. The resulting second-order differential equation is:

$$\frac{d^2\theta}{dt^2} = -\frac{g}{l}\sin(\theta) - \frac{b}{m}\frac{d\theta}{dt} + \frac{1}{ml}f_e(t), \tag{17.8}$$

where b is the coefficient of dynamic friction, m is the mass of the pendulum, and $f_e(t)$ is a time-dependent external driving force. We will use, in particular, a sinusoidal driving force of the form $f_e(t) = A\sin(Ft)$, where A and F are the amplitude and frequency of this force, respectively.

2. We will modify the view so that it displays a phase-space diagram of the system, that is, a plot of angular position versus angular velocity.

3. We will modify both the model and the view to compute and plot the potential and kinetic energies of the system and their sum.

4. We will show how to modify the introduction pages to update the description of the simulation.

Modifying the Model

We need to revisit the different subpanels for the model and make the neccessary changes to each of them.

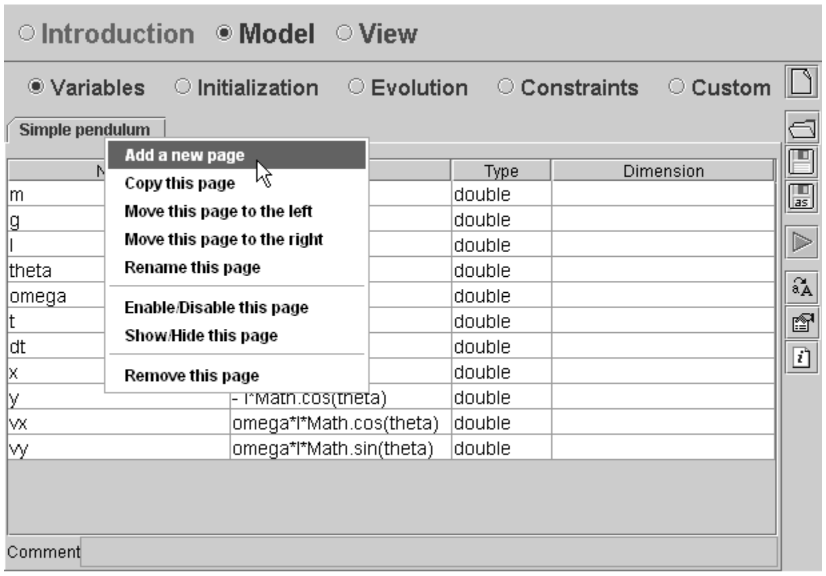
Adding New Variables

The introduction of friction and an external driving force requires adding new variables to the model. We do this by creating a second table of variables. Although we could add the new variables to the existing table, it is sometimes preferable, for clarity, to organize the variables into separate tables. For this, we select the `Variables` subpanel of the `Model` panel of *Ejs* and right-click on the upper tab of the existing page. A popup menu will appear as shown in Figure 17.16.

From this menu, we select the `Add a new page` option (the one highlighted in the figure), and *Ejs* will create a page with an empty table of variables. (Before creating it, though, *Ejs* will ask you for the name of this new page. You can choose, for instance, `Damping, forcing, and energy`.)

In this new table we can type all the new variables that help us extend the model in the prescribed way. The mechanism to add a variable is simple. We just need to type a name for the variable in the column `Name`, select one of the possible types in the `Type` column, and, optionally, provide an initial value in the `Value` column. (The column labeled `Dimension` is used to declare arrays, and we won't use it for this model.)

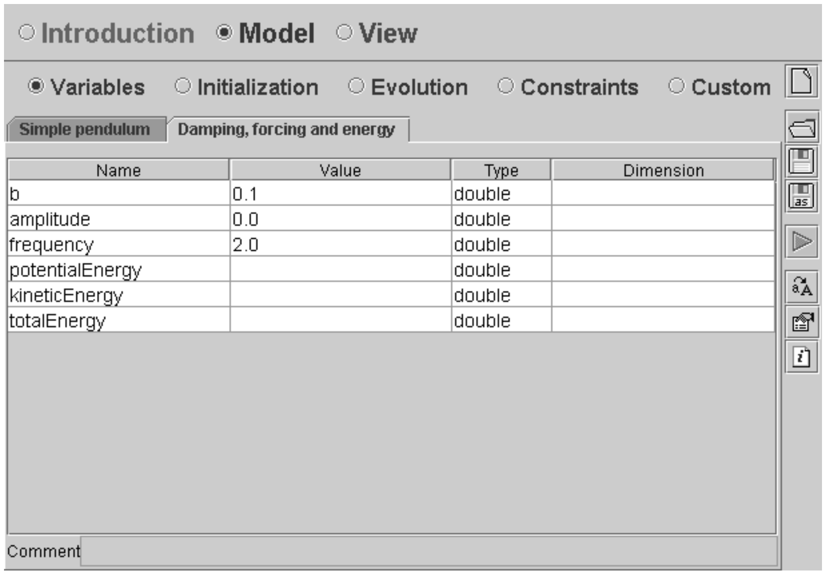
Double-click a cell in the table in order to edit that cell. Use the tab key or arrow keys to move from cell to cell. *Ejs* will automatically add rows as needed. In this way, create new variables of type `double` called `b`, `amplitude`, `frequency`, `potentialEnergy`, `kineticEnergy`, and `totalEnergy`. Assign the first three variables the initial values of `0.1`, `0.0`, and `2.0`, respectively. The energy variables will be initialized as result of the evaluation of constraints. The final new



N		Type	Dimension
m		double	
g		double	
l		double	
theta		double	
omega		double	
t		double	
dt		double	
x		double	
y	- l*Math.cos(theta)	double	
vx	omega*l*Math.cos(theta)	double	
vy	omega*l*Math.sin(theta)	double	

FIGURE 17.16 The popup menu for a page of variables.

table of variables is displayed in Figure 17.17. Note that *Ejs* will add an empty row to the end of the table. This can be deleted by right-clicking on the row and selecting `Remove this variable`.



Name	Value	Type	Dimension
b	0.1	double	
amplitude	0.0	double	
frequency	2.0	double	
potentialEnergy		double	
kineticEnergy		double	
totalEnergy		double	

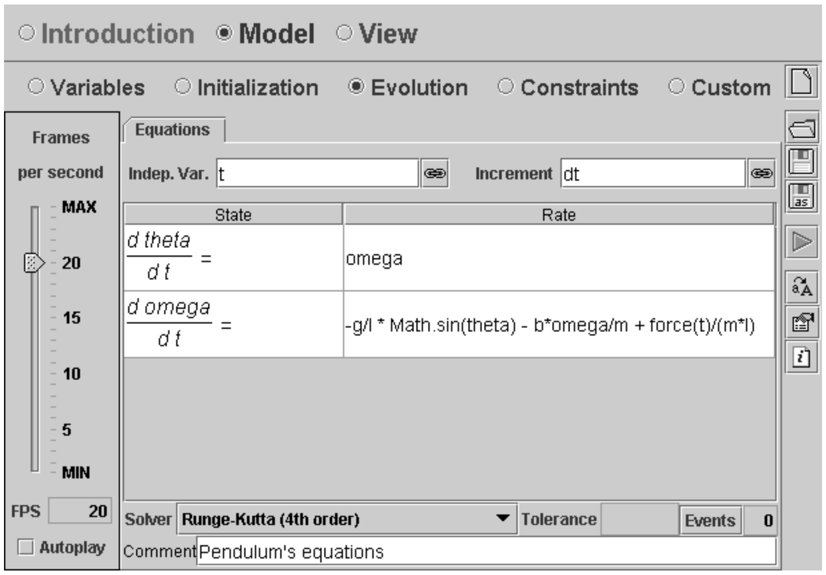
FIGURE 17.17 The new table of variables for our simulation.

Modifying the Evolution

We need to edit the differential equations for the system to add the new forces. For this, go to the Evolution subpanel and edit the right-hand side of the second differential equation so that it reads:

```
-g/l * Math.sin(theta) - b*omega/m + force(t)/(m*l)
```

The result is shown in Figure 17.18. Notice that we are using in this expression the method force(t) that is not yet defined. We will need to create it as a user-defined method, when we get to the Custom subpanel.



Indep. Var. t Increment dt

State	Rate
$\frac{d\,theta}{d\,t}$ =	omega
$\frac{d\,omega}{d\,t}$ =	-g/l * Math.sin(theta) - b*omega/m + force(t)/(m*l)

FPS 20 Solver Runge-Kutta (4th order) Tolerance Events 0

Comment Pendulum's equations

FIGURE 17.18 The edited differential equations.

Computing the Energy

Select the Constraints subpanel and follow a procedure similar to what we did for the table of variables to create a second page of constraints. Call the new page Energies and, in the blank editor that appears, type the following code:

```
potentialEnergy = m*g*(y+l);
kineticEnergy = 0.5*m*(vx*vx+vy*vy);
totalEnergy = potentialEnergy + kineticEnergy;
```

(This sets the potential energy to zero when the pendulum is hanging straight down.) The reasons to compute the energies in a page of constraints (instead of in the evolution) are the same as those for the computation of the cartesian magnitudes explained in Section 17.3. Any expression that we specify as constraints will be evaluated every time the state of the system changes. Thus, the corresponding relationships (the value of the energy variables) are always kept up to date.

Coding the external force

To finish our changes to the model, we need to specify the expression for the external force. We do this using a page of `Custom` code. Move to this subpanel and click in the empty work area to create a new page called `External force`. The new page that appears looks very much like the other editors of code that we have used previously. But there is an important difference.

Since custom code is not automatically used by *Ejs*, the code we write here is not wrapped into an internal Java method, but must be explicitly defined as a valid Java method (and later invoked in your code). For this reason, type in the editor exactly the following:

```
public double force (double time) {
  return amplitude * Math.sin(frequency*time);
}
```

This correctly defines the custom method, and concludes our changes to the model.

Modifying the View

Let us start the changes to the view of the simulation by including the phase-space graph of angular velocity versus angular position. For this, go to the `View` panel and, from the right-hand side collection of elements offered by *Ejs*, click on the icon for a `PlottingPanel`, .

When you click on it, the icon will be highlighted, and the cursor will change to a magic wand, . With this wand, go to the left-hand side frame of the view, and, in the tree of elements, click on the element called `Dialog`. You are then asking *Ejs* to create a new plotting panel as child of the `Dialog` window. This is the simple mechanism used to add new elements to the view of the simulation. Figure 17.19 illustrates this action.

When you do this (and after providing a name for the new element), a new plotting panel will appear in the dialog window, sharing the available space with the previous plotting panel. Because the `Dialog` window is too small to host both panels, we need to enlarge it. Get rid of the magic wand by clicking on any blank area of the left frame. Double-click the `Dialog` element and set its `Size` property to `372,600`. You can also edit the properties of the new plotting panel to customize its title and axis labels by double-clicking on the new plotting panel element that you created.

Now, let's add a `Trace` element to the new plotting panel. A trace is an element that can display a graph consisting of a sequence of points. Follow the creation process described above (again using the magic wand), name the new element `PhaseSpace`, and edit the table of properties so that it looks like Figure 17.20. These properties simply instruct the element to add to the graph a new

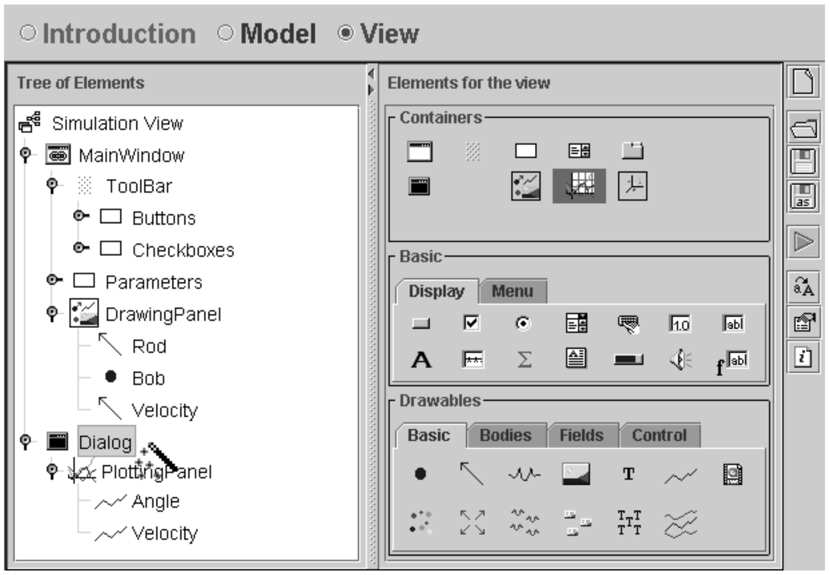
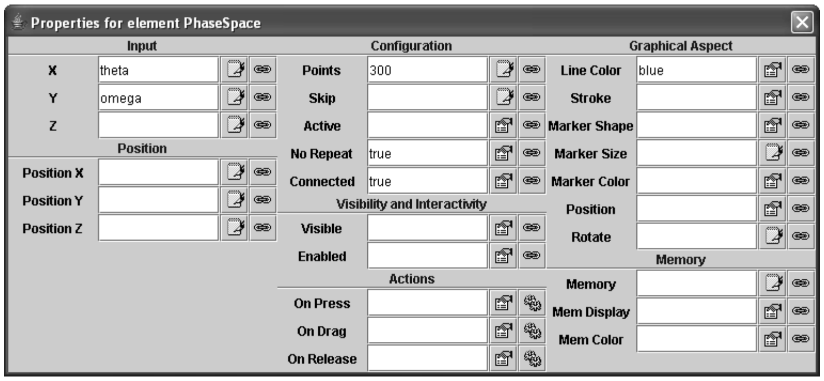


FIGURE 17.19 Adding a new plotting panel element to the `Dialog` window.

point (θ, ω) after each evolution step, displaying only the last 300 added points, and connecting them with a blue line.


Properties for element PhaseSpace

Input
X: theta
Y: omega
Z:

Position
Position X:
Position Y:
Position Z:

Configuration
Points: 300
Skip:
Active:
No Repeat: true
Connected: true

Visibility and Interactivity
Visible:
Enabled:

Actions
On Press:
On Drag:
On Release:

Graphical Aspect
Line Color: blue
Stroke:
Marker Shape:
Marker Size:
Marker Color:
Position:
Rotate:

Memory
Memory:
Mem Display:
Mem Color:

FIGURE 17.20 Properties for the `PhaseSpace` trace element.

The changes we did so far illustrate very well how to add and customize new elements to the view. It is as simple as it looks. Just use the magic wand to add new elements, and edit their properties to match your needs. In most cases, you will use some of the variables of the model for the properties of the view element. The connection between model and view is then automatically handled by *Easy Java Simulations*.

If we now run the simulation, we would obtain something like Figure 17.21.

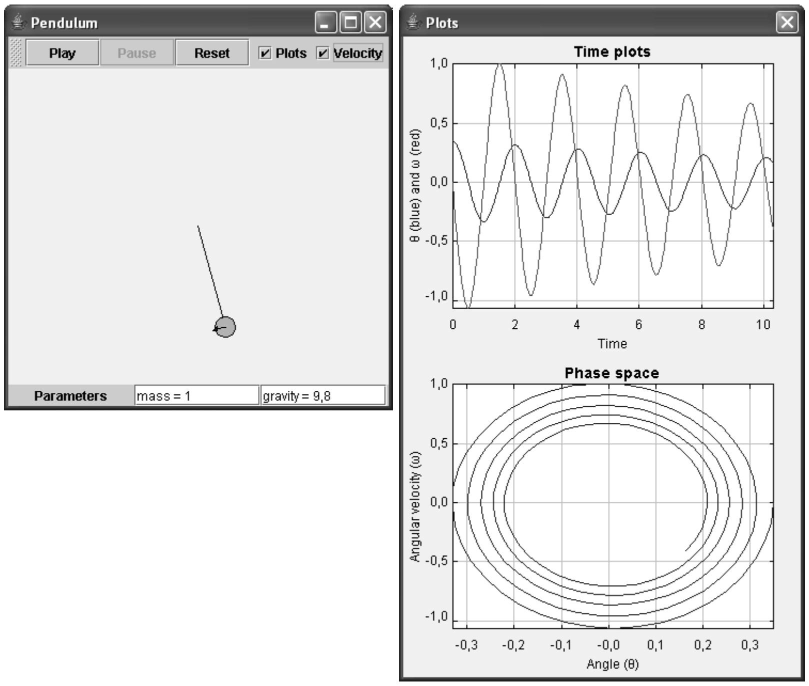


FIGURE 17.21 The simulation displaying both time and phase space plots.

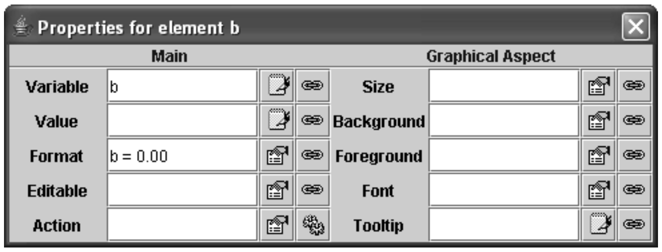
We can now proceed similarly to add time plots of the energies of the system. We leave it to you as an exercise to create a third plotting panel as a child of the `Dialog` window and three traces (with different colors) in this panel that will plot the potential, kinetic and total energies of the system.

We will end the changes to the view by adding new fields for the user to visualize and edit the values of the variables `b`, `amplitude` and `frequency`. Click on the icon for view elements of type `NumberField`, 1.0, and add three of these to the view element called `Parameters`. Give the three fields the same name as the corresponding variables, that is, `b`, `amplitude`, and `frequency`. (Names of elements must be unique in the view, but they don't clash with the name of model variables.)

`Parameters` is a basic Swing [7] panel that has been configured (using its `Layout` property) to display its children using a grid with one single row. When we add the new three fields, the six children of the panel will look pretty small. To solve this, change the `Layout` property of `Parameters` to `grid:2,3`. This will organize the children in two rows of three elements each.

[7]Swing is the standard Java library for graphical components such as panels, buttons, and labels.

Now, we need to edit the table of properties of each of the field elements so that they display the corresponding variables. We show how to do this for the first element. Double-click the element `b` and edit its properties as shown in Figure 17.22.



Properties for element b

	Main		Graphical Aspect
Variable	b	Size	
Value		Background	
Format	b = 0.00	Foreground	
Editable		Font	
Action		Tooltip	

FIGURE 17.22 Properties for the `b` field element.

The association of the property `Variable` with the variable `b` of the model tells the element that the value displayed or edited in this field is that of `b`.

> To facilitate the association of properties with variables of the model, you can use the icon 🔗 that appears to the right of the text field for the corresponding property. If you click on this icon, a list of all the variables of the model that can be associated to this property will be offered to you. You can then comfortably select the one you want with the mouse.
> Also, to help edit some of the technical properties (such as layouts, colors, and fonts, for instance), *Ejs* offers dedicated editors that can be accessed clicking on the icon 🗎, when shown.

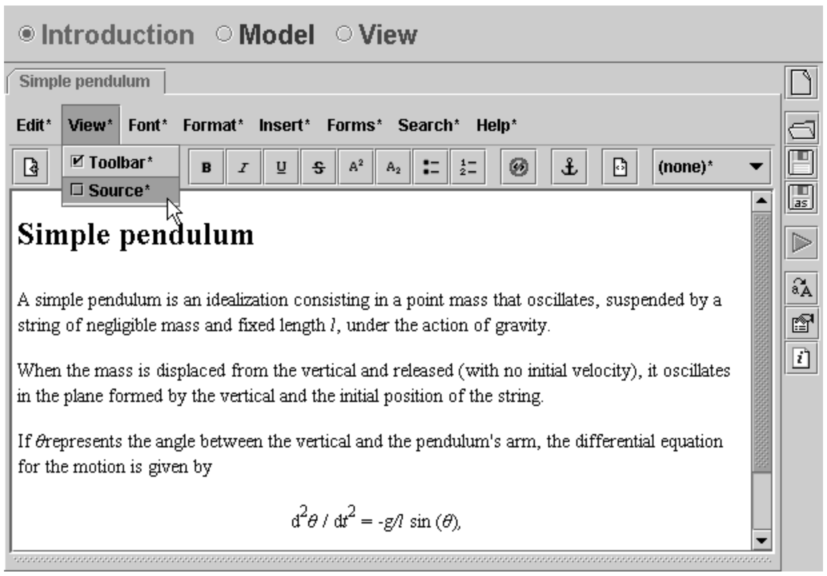
The property called `Format`, to which we assigned the value `b = 0.00`, has a special meaning. It doesn't indicate that `b` should take the value of 0, but is interpreted by the element as an instruction to display the value of `b` with the prefix "b = " and with two decimal digits.

This completes our changes to the view.

Modifying the Introduction

We now want to modify the introduction to reflect the new situation. Select the `Introduction` panel of *Ejs* and right-click on the tab of the existing page to bring in its popup menu. From this menu, select the `Edit/View this page` option. This will activate the edit mode for the displayed html page. See Figure 17.23.

HTML pages are text pages that include special instructions or tags that allow web browsers to give a nice format to the text, as well as to include several types of multimedia elements. The editor allows you (when in edit mode) to work in a WYSIWYG (what you see is what you get) mode. However, if you are familiar with html and want to work directly on the code (which, sometimes, is preferable), you can select the option highlighted in Figure 17.23 to access directly the html source for the page.



Simple pendulum

A simple pendulum is an idealization consisting in a point mass that oscillates, suspended by a string of negligible mass and fixed length *l*, under the action of gravity.

When the mass is displaced from the vertical and released (with no initial velocity), it oscillates in the plane formed by the vertical and the initial position of the string.

If θ represents the angle between the vertical and the pendulum's arm, the differential equation for the motion is given by

$$d^2\theta \,/\, dt^2 = -g/l \sin(\theta),$$

FIGURE 17.23 Edit mode of the html editor for the introduction page.

You can write as many introduction pages as you want. As we saw earlier, each of the pages will turn into a link in the main html page that *Ejs* generates for the simulation. To return to View mode, right-click the tab of the existing page and select again `Edit/View this page`. In this way, you toggle the edit/view mode.

An Improved Laboratory

Our new simulation is finished. We need now to save it to disk. Because we don't want to loose our original simulation, click on the `Save As` icon of *Ejs*'s taskbar, and a file dialog will allow you to save the simulation to a new file.

This simulation allows you to explore the behavior of a simple pendulum playing with different possible values of the parameters. Running the simulation can produce situations such as that displayed in Figure 17.1, at the beginning of this Chapter.

You will find the complete improved simulation in the **_ospguide** subdirectory of your **Simulations** directory with the name **PendulumComplete.xml**.

17.6 ■ A GLOBAL VISION

We end this chapter with an overview of what we did. We learned that *Easy Java Simulations* is an authoring tool that provides a simplified way to design and build complete interactive simulations in Java. For this, it structures a simulation into three main parts, and provides, for each of these parts, a specialized editor that helps you implement them using high-level access to many of the OSP classes and utilities.

To investigate in more detail how each of these editors work, we loaded an existing simulation and inspected all of the panels and subpanels of *Ejs* in turn. This helped us understand how the different parts of the simulation are specified and how all the pieces fit together.

We also learned how to run and distribute the simulation either using physical media or through the Internet. Finally, we modified the simulation in order to learn some of the operating procedures of *Easy Java Simulations*.

The three parts of a simulation are the introduction, the model and the view. Each of these parts has its function in the simulation and its own panel in *Ejs*' interface, each with its own look and feel.

The introduction offers an editor for the html pages required to create the multimedia narrative that introduces the simulation. This editor allows us to edit this narrative, either working in WYSIWYG mode or writing directly the html code.

The model is the engine of the simulation and is created using a sequence of subpanels where we specify the different parts of it: definition of variables, initialization, evolution of the system, constraints (or relationships among variables) and custom methods. Each panel provides editing tools that facilitate the job of creation (including sophisticated tasks such as solving differential equations and treatment of events).

Finally, the view contains a set of predefined elements, based on Swing components and OSP graphics classes, that can be used as individual building blocks to construct a structure in form of a tree for the interface of our simulation. These elements, that can be added to a view through a simple procedure of click and create (our magic wand), have in turn a set of properties that indicate how each element looks and behaves. These properties, when associated to variables from the model (or Java expressions that use them), turn the simulation into a true dynamic and interactive visualization of the phenomenon under study.

And that's it! Though the chapter is long because we accompanied the description with details and instructions, the process can be summarized in the few paragraphs above. Obviously, learning to manipulate the interface of *Easy Java Simulations* with fluency requires a bit of practice, as well as the familiarization with all the possibilities that exist. In particular, with respect to the creation of the view, you'll need to learn the many types of elements offered and what each of them can do for you.

If you want to know more about *Easy Java Simulations* or want to see more examples of simulations created with *Ejs*, you are cordially invited to read the manual found in the companion CDROM and to visit *Ejs*'s home page at http://fem.um.es/Ejs.