Chapter 2

Creating Simulations

Adapted from *An Introduction to Computer Simulation Methods* by Harvey Gould, Jan Tobochnik, and Wolfgang Christian

We introduce Java and *EJS* in the context of simulating the motion of falling particles near the Earth's surface using simple Java syntax and *EJS* user interface elements. A simple algorithm for solving first-order differential equations numerically also is discussed.

2.1 Introduction to modeling

If you were to take a laboratory-based course in physics, you would soon be introduced to the oscilloscope. You would start by learning the function of many of the knobs, how to read the display, and how to connect various devices so that you could measure various quantities. If you did not know already, you would learn about voltage, current, impedance, and AC and DC signals. Your goal would be to learn how to use the oscilloscope to analyze circuits and in so doing you are learning about the inner workings of the oscilloscope. You may end up only using the oscilloscope as a laboratory tool, but if you continue your electronics and instrumentation training, you may some day contribute to building a better oscilloscope.

The same approach can be easily adopted with a simulation tool such as *EJS*. You will learn how to implement algorithms to model physical systems within *EJS* and in so doing you will begin to learn about Java. If you continue to study programming, you may enhance *EJS* or contribute to its development. In this chapter, we will present some of the essential Java syntax and introduce *EJS elements*, which will allow us to create simulations with a graphical user interface and visual output such as plots.

One of the ways that science progresses is by making models. If the model is sufficiently detailed, we can determine its behavior, and then compare the behavior with experiment. This comparison might lead to verification of the model, changes in the model, and further simulations and experiments. In the context of computer simulation, we usually begin with a set of initial conditions, we compute the dynamical behavior of the model, and we visualize the data in the form of tables of numbers, plots, and animations. We begin with a simple example to see how this process works.

Imagine a particle such as a ball near the surface of the Earth subject to a single force, the force of gravity. We assume that air friction is negligible and the gravitational force is given by

$$F_g = -mg, \tag{2.1}$$

where m is the mass of the ball and $g = 9.8\,\mathrm{N/kg}$ is the gravitational field (force per unit mass) near the Earth's surface. To make our example as simple as possible, we first assume that there is only vertical motion. We use Newton's second law to find the motion of the ball,

$$m\frac{d^2y}{dt^2} = F, \tag{2.2}$$

where y is the vertical coordinate defined so that up is positive, t is the time, F is the total force on the ball, and m is the inertial mass (which is the same as the gravitational mass in (2.1)). If we set $F = F_g$, (2.1) and (2.2) lead to

$$\frac{d^2y}{dt^2} = -g. \tag{2.3}$$

Equation (2.3) is a statement of a model for the motion of the ball. In this case the model is in the form of a second-order differential equation.

You are probably familiar with the model summarized in (2.3) and know the analytical solution:

$$y(t) = y(0) + v(0)t - \frac{1}{2}gt^2 \tag{2.4a}$$

$$v(t) = v(0) - gt. \tag{2.4b}$$

We will determine the motion of a freely falling particle numerically to introduce the tools that we will need in a familiar context.

We begin by expressing (2.3) as two first-order differential equations:

$$\frac{dy}{dt} = v \tag{2.5a}$$

$$\frac{dv}{dt} = -g, \tag{2.5b}$$

where v is the vertical velocity of the ball. We next approximate the derivatives by small (finite) differences:

$$\frac{y(t+\Delta t) - y(t)}{\Delta t} = v(t) \tag{2.6a}$$

$$\frac{v(t+\Delta t) - v(t)}{\Delta t} = -g. \tag{2.6b}$$

Note that in the limit $\Delta t \to 0$, (2.6) reduces to (2.5). We can rewrite (2.6) as

$$y(t+\Delta t) = y(t) + v(t)\Delta t \tag{2.7a}$$

$$v(t+\Delta t) = v(t) - g\Delta t, \tag{2.7b}$$

The finite difference approximation we used to obtain (2.7) is an example of the *Euler* algorithm. Equation (2.7) is an example of a *finite difference* equation, and Δt is the time step.

Now we are ready to follow $y(t)$ and $v(t)$ in time. We begin with an initial value for y and v and then *iterate* (2.7). If Δt is sufficiently small, we will obtain a numerical answer that is close to the solution of the original differential equations in (2.6). In this case we know the answer, and we can test our numerical results directly. Before building this free fall model, we examine and run an existing model in Exercise 2.1 without concerning ourselves with the implementation details. (See Appendix 1A for a tutorial that shows how to examine and run an existing *EJS* model.)

Exercise 2.1. A first-order ODE example

Consider the first-order differential equation

$$\frac{dy}{dx} = f(x), \tag{2.8}$$

where $f(x)$ is a function of x. The approximate solution as given by the Euler algorithm is

$$y_{n+1} = y_n + f(x_n)\Delta x. \tag{2.9}$$

This algorithm is implemented in the Euler Method model and its xml source code is the `EulerMethod.xml` file in the Chapter 2 source code examples directory. Copy the Chapter 2 examples into your *EJS* workspace as described in Section 2.7. Examine the model in *EJS* and note that the rate of change of y on the Evolution workpanel has been approximated by its value at the *beginning* of the interval, $f(x_n)$.

a. Suppose that $f(x) = 2x$ and $y(x = 0) = 0$. The analytical solution is $y(x) = x^2$, which we can confirm by taking the derivative of $y(x)$. Convert (2.8) into a finite difference equation using the Euler algorithm. For simplicity, choose $\Delta x = 0.1$. It would be a good idea to first use a calculator or pencil and paper to determine y_n for the first several time steps.

b. Sketch the difference between the exact solution and the approximate solution given by the Euler algorithm. What condition would the rate of change, $f(x)$, have to satisfy for the Euler algorithm to give the exact answer?

■

One of the advantages of *EJS* is that little explicit code is written to produce a running model. The Evolution workpanel and the button and input field actions contain less than a dozen statements. The heart of the model is the code on the Evolution workpanel.

```
y= y  + _view.rateFunction.evaluate(x)*dx; // advances y
x = x + dx;                                // advances x
```

This code advances the differential equation by evaluating the rate function df/dt that was entered into the user interface. The details of how to read user keyboard input and interpret this input as a mathematical expression are unimportant.[1] *EJS* provides the tools that make the implementation easy. Navigate to the View workpanel and examine the button and field property inspectors by double-clicking on the elements in the tree of elements to do Exercise 2.2.

[1]See Appendix 2A for information about the *EJS* parser.

Exercise 2.2. Element actions

Describe the button and field actions in the Euler Method model. In other words, examine the button and text field inspectors by double clicking on the elements in the tree of elements and describe the actions that are performed? ■

Problem 2.1. Invent your own numerical algorithm

As we have mentioned, the Euler algorithm evaluates the rate of change of y by its value at the beginning of the interval, $f(x_n)$. The choice of where to approximate the rate of change of y during the interval from x to $x + \Delta x$ is arbitrary, although we will learn that some choices are better than others. All that is required is that the finite difference equation must reduce to the original differential equation in the limit $\Delta x \to 0$. You can, for example, advance the independent variable before computing the rate. Does this produce a better solution? Think of several other algorithms that are consistent with the small step size limit and test your algorithms. ■

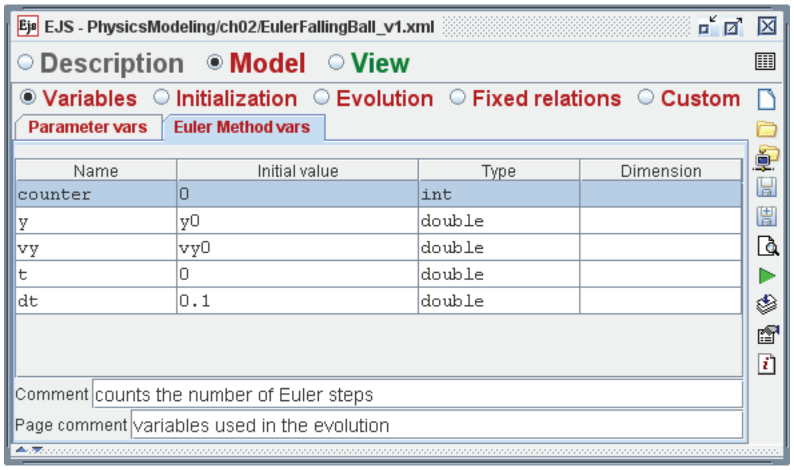
2.2 Modeling free fall

Because we have a good idea how such a model should behave, we start by build a model to numerically solve for the position and velocity of a falling particle. A particle is a point mass without structure or internal degrees of freedom and we will represent it using a ball when we add a visualization to our model. The first version of our Euler Falling Ball model is defined in a file named `EulerFallingBall.xml` in the `PhysicsModeling` subdirectory of the *EJS* workspace source directory. Open this model in *EJS* and note that it consists of a documentation page, a variables page, and an evolution page with Java code. [2] This model is designed to teach Java syntax and does not have a user interface. It is always a good idea to save a copy of an example model in a working subdirectory within the source directory of the *EJS* workspace as you work the chapter exercises. You can do so using the save as a new file icon on the *EJS* toolbar.

Navigate to the Variables workpanel in the Euler Falling Ball model and examine the tables of variables shown in Figure 2.1. The first table defines the initial conditions and the second table defines variables that are modified during the model's evolution. The `counter` variable is declared to be an integer because whole number arithmetic is fast and does not have any roundoff error. The model's position and velocity variables are double precision (approximately 16 significant digits) numbers with their usual physical meaning as described in the Comment field at the bottom of the workpanel. Note how variables defined in the first table are used to set the values of variables in the second table.

The Evolution workpanel implements the Euler algorithm given in (2.7). The evolution step computes the numerical and analytic solutions and prints the results. We will next describe the syntax used in each line on this code page.

[2] *EJS* uses the information in the model's xml file to create Java source code that is compiled into *byte code* and then executed. The source code is in the *EJS* workspace source directory and Java compiler places the byte code into a jar file in the workspace output directory. The jar file has the name of the xml file but with the extension `jar`. A jar file is a *Java archive* that is similar to a zip file and can be opened with any standard programs such as *7-Zip* on Windows or *StuffIt* on the Mac. Use one of these archive tools to examine the compiled code from the first model in the EulerFallingBallApp.jar file in the output directory. The jar file in the output directory contains only the code for the given model and is not a stand-alone ready to run program because it does not contain the *OSP* and *EJS* libraries. We explain how to create a ready-to-run jar file later in this chapter.



Name	Initial value	Type	Dimension
counter	0	int	
y	y0	double	
vy	vy0	double	
t	0	double	
dt	0.1	double	

Comment: counts the number of Euler steps

Page comment: variables used in the evolution

Figure 2.1: The Euler Falling Ball model has two variable tables and the variables defined in these tables are used on the Evolution workpanel. The *EJS* toolbar on the right is used to load, run, and save models. Right-click on the toolbar icons to see all toolbar actions.

Listing 2.1: First version of a falling particle model.

```
counter++;                                               // increment counter
y = y + vy*dt;                                           // advance position
vy = vy -g*dt;                                           // advance velocity
t = t + dt;                                              // advance time
double yAnalytic=y0+vy0*t-0.5*g*t*t;                     // analytic y
double vyAnalytic=vy0-g*t;                               // analytic vy
// print results
_println("counter="+ counter+ "  time="+t);              // counter and time
_println("Euler: y="+y+ "  vy="+vy);                     // Euler
_println("Analytic: y="+yAnalytic+" vy="+vyAnalytic);// analytic
_println();                                              // blank line
if(y<0) _pause();                                        // pause simulation
```

Java code consists of a sequence of *statements* that tell the program to do something and each statement ends with a semicolon. The first line in Listing 2.1 is a statement that increments the counter using the Java `++` operator. (See Appendix 2B for a description of Java operators.) The second, third, and fourth lines advance the dynamical variables using Euler's algorithm. For example, the second line multiplies the values of `vy` and `dt`, adds the result to the variable `y`, and stores the result in `y`.

```
y = y + vy*dt;    // advance position
```

A double slash begins an end of line comment and the *EJS* code editor shows these comments using a red font. Code comments are used to describe the technical details of a model and are ignored by the Java compiler. Words that cannot be used for variables because they are native to the Java language, such as `double` and `if`, are reserved and are shown using a blue font. Java reserves fifty four words so the language itself is very compact. You can change the code editor's font size by selecting the Edit options icon on the taskbar and selecting the Change font option on the Aspect tab.

The advance position statement is similar to mathematics but there are important differences. Variables such as `y` and `vy` represent memory locations and are finite precision approximations to real numbers. More importantly, the equal sign (=) is not a statement of equality. It is an assignment operator that replaces what is stored memory. In this example, the memory location storing the current `y` value is updated with the result of the computation to the right. If the statement were interpreted mathematically, then we could cancel the y variable and erroneously conclude that $v_y dt = 0$.

Lines five and six of Listing 2.1 declare and initialize `yAnalytic` and `vyAnalytic` variables for the analytic solution. Variables such as integer, floating point, boolean, and character variables are *primitive data types* and are easy to create and access as they are needed. We specify the type of variable followed by a meaningful name. There are four types of integers (`byte`, `short`, `int`, and `long`) and two types of floating point numbers (`float` and `double`); the differences are the range of numbers that these types can store. We will almost always use type `int` because it does not require as much memory as type `long`. There are two types of floating point numbers, but we will use type `double`, the type with greater precision, to minimize roundoff error and to avoid having to provide multiple versions of various algorithms.

A variable must be *declared* before it can be used and it can be initialized at the same time that its type is declared. If a variable is declared but not initialized, for example,

```
double yAnalytic;
```

then the variable is assigned a default value. It is a good idea to initialize all variables explicitly and not rely on their default values. Declaration and initialization can be done in a single statement as in Listing 2.1 or using an entry in an *EJS* Variables table but there is an important difference. The *scope* of a variable determines where it can be used. Variables defined in an *EJS* Variables table are global and can be used anywhere within the model. Variables defined on a code page are local and can only be used on that page.

Lines eight through eleven display the results of our computation using the `_println` method. A *method* is a block of Java code that can be *called* (invoked) to do something without our needing to know exactly how it is done. The `_println` method, for example, prints text in the *EJS* Console and we usually don't care how the code was written so long as it works. The parameter passed to this method, which appears between the parentheses, is a `String`. A `String` is a sequence of characters and can be created by enclosing text in quotation marks as shown in Listing 2.1.

We can create and assign a value to a String variable as we would number variables but there are important differences. A String is composed of characters and the amount a memory used by a String is not fixed but is determined by the number of characters. Furthermore, String variables can give us information about themselves and can manipulate themselves using the *dot* operator as shown in the following code fragment:

```
String message="Old men are fun.";  // creates String
int n=message.length();              // counts characters; n=16
message=message.substring(4);        // new String starting at character 4
message=message.replace('m','M');    // new String with character replaced
_println(message);                   // print string "Men are fun."
```

Variables, such as String variables, that are composed of other variables and that have methods that act on their internal data are know as compound variables or *objects*.

Exercise 2.3. Spoonerism Spoonerisms are words or phrases in which words or syllables get swapped. Write code to swap the first letters in "bad salad" and print the result. ■

We displayed our numerical results in Listing 2.1 by using the `+` operator. When applied to a `String` and a number, the number is converted to the appropriate `String` and the two Strings are concatenated (joined). This use is shown in the three `_println` statements in lines 21 through 29 of Listing 2.1. Note the different outputs produced by the following two statements:

```
_println(("x = " + 2) + 3);     // displays x = 23
_println("x = " + (2 + 3));     // displays x = 5
```

The parentheses in the second line tell the compiler to interpret the `+` operator as an addition operator, but both `+` operators in the first line are treated as concatenation operators.

Line twelve uses an `if` statement to make a decision. If the boolean expression following the `if` evaluates to `true` the statement executes the following expression. In our model the `_pause()` method is invoked if the y-position of the particle is less than zero because the expression `(n<0)` is true. The `_pause()` method is another example of a predefined *EJS* method that does something behind the scene.[3]

EJS is designed to make it easy to model the time evolution of dynamical systems and the Evolution workpanel does just that. Code that is placed here is executed by a hidden timer at a rate that is determined by the frames per second `FPS` parameter on the Evolution workpanel. Our model's simulation time advances by 0.1 units two times per second because the frames per second parameter is two. We use a slow simulation rate so that we can observe the data as it is being generated. The model's evolution begins when the run button (green triangle) is pressed. The timer starts automatically because the Autoplay property is checked. The timer stops and the evolution ends when the ball's position is less than zero but stopping the evolution does not stop (exit) the program. You must exit the program by right-clicking on the run button (red triangle) and choose kill the current simulation.

Test your understanding of *EJS* by doing Exercises 2.4 and 2.5.

Exercise 2.4. Free Fall Error

Add a print statement to the Euler Falling Ball model to display the error between the numerical and the analytic solutions. Show that the position error increases linearly with time. Show that the velocity error remains zero. Why does Euler's method only have a position error for this problem? ■

Exercise 2.5. Summing a series

Create a model to sum the following series for a given value of N:

$$S = \sum_{m=1}^{N} \frac{1}{m^2}. \tag{2.10}$$

[3]The `_println` and `_pause` methods used in Listing 2.1 are not standard Java methods and can only be used within *EJS*. Because internal *EJS* methods and variables begin with the underscore character, users should not define methods or variables that begin with the underscore special character.

Start from scratch by clicking on the create a new model button 🗋 on the *EJS* toolbar and saving it in subdirectory within the source directory of the *EJS* workspace. Create a variables table and then create an evolution page that adds terms to the sum. The evolution can print the series as each term is added.

a. First run your model with $N = 10$. Then run for larger values of N. Does the series converge as $N \to \infty$? What value of N is needed to obtain S to within two decimal places?

b. Modify your model so that the summation continues until the added term to the sum is less than some value ϵ. Run your program for $\epsilon = 10^{-2}$, 10^{-3}, and 10^{-6}.

c. Instead of using the `=` operator in the statement

```
sum = sum + 1.0/(m*m);
```

use the equivalent operator:

```
sum += 1.0/(m*m);
```

Check that you obtain the same results. Operators are described in Appendix 2B.

■

Caution: A common programming pitfall when implementing the series model is to program the summation using `1/(m*m)`. Almost all programming languages will evaluate this express as zero because the division is performed using integer arithmetic. Integer division produces an integer result and the reminder is lost. Integer arithmetic is used if both arithmetic operands are integers. Otherwise floating point arithmetic is used. As with mathematical operations, computational expressions evaluate left to right while giving precedence to terms within parenthesis.

What if we wish to generate data as fast as possible without a timer? We could, of course, set the steps per display (SPD) parameter in the evolution very high but another approach is to not use a timer at all but to do the computation the moment that the program starts. Reopen the Euler Falling Ball model and navigate to the Evolution workpanel. Right click on the tab for this code page and disable the page. Note how the tab name is changed. Navigate to the initialization page, click within the workpanel to create a code page, and enter the following code:

Listing 2.2: Second version of a falling particle model.

```
while(y>=0){
  counter++;                                            // increment counter
  y = y + vy*dt;                                        // advance position
  vy = vy -g*dt;                                        // advance velocity
  t = t + dt;                                           // advance time
  double yAnalytic=y0+vy0*t-0.5*g*t*t;                  // analytic y
  double vyAnalytic=vy0-g*t;                            // analytic vy
  // print results
  _println("counter="+ counter+ "  time="+t);           // counter and time
  _println("Euler: y="+y+ "  vy="+vy);                  // Euler
  _println("Analytic: y="+yAnalytic+" vy="+vyAnalytic);// analytic
  _println();                                           // blank line
}
```

Save this new version of the Euler Falling Ball model in your workspace.

The second version of our model uses a `while` statement to repeatedly execute a block of code.

```
while (boolean-expression) {
   // statements go here
}
```

The `while` statement begins by evaluating a boolean expression. If this expression is true, then the statement following the boolean expression is executed. Otherwise the statement is skipped. We use curly braces to group multiple statements so that they are treated as a single statement. Note that we indent the loop's code to improve readability. You can right-click within a code page and select the format code option to automatically perform this indentation. Formatting often finds syntax errors such as unbalanced braces.

In addition to code formatting, the *EJS* code editor has other features to help produce syntactically correct code. Reserved words (keywords) that are part of the Java language and cannot be used as variable names are shown using a light blue font color. Comments are shown using a red font color. Right-clicking and selecting the open code wizard displays a dialog that inserts statement templates into the code page. You can, for example, insert a `do/while` statement to test a condition after the body of the loop has been executed:

```
do {
   // statements go here
}while (boolean-expression);
```

A `for` loop is often used to repeat a statement a countable number of times.

```
for(int i=0; i<n; i++) {
   // statements go here
}
```

The `for` statement defines and initializes a counter (`int i=0`) and tests a condition to determine if the body of the loop should execute (`i<n`). If the condition is true, the body of the loop is executed and the counter is updated (`i++`) before again testing the condition to determine if execution should continue. We use this looping pattern in Exercise 2.6.

Exercise 2.6. Using a for loop

Rebuild the model that was used in Exercise 2.5 to use a for loop rather than a timer. Print only the result of the summation. The following statements may be useful:

```
double sum = 0;          // sum is equivalent to S in (2.9)
for (int m = 1; m <= N; m++) {
   sum = sum + 1.0/(m*m); // loop body
}
```

Note that in this case it is more convenient to start the loop from $m = 1$ instead of $m = 0$ to avoid dividing by zero. ■

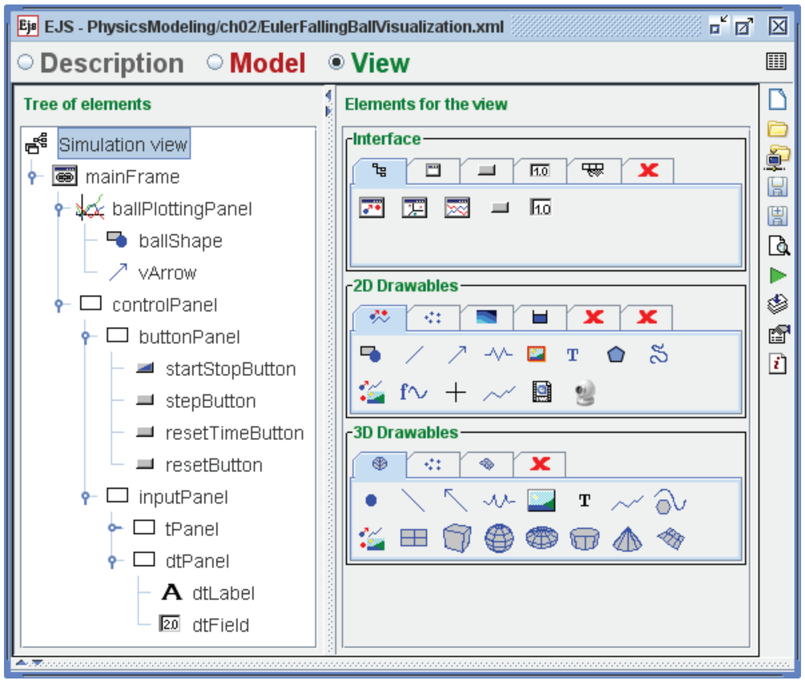


Figure 2.2: The View shows the structure of a model in its tree of elements. Elements are added to the View by clicking on a palette icon and then clicking within the tree.

2.3 Visualizing free fall

You might have found that doing the analysis for Exercises 2.4 and 2.5 was tedious. It would be desirable if we could intervene while the model was running and be able to visualize the results. For example, in Exercise 2.4 we might want to change the value of Δt so that we could see how this value affects the error. For Exercise 2.5 we might want to control the number of terms in the summation without editing the variables table and recompiling the model. *EJS* makes it easy to build graphical user interfaces to perform these tasks and this section describes how this is done. The model we are about to build can be found in the Euler Falling Ball Visualization model in the Chapter 2 source code directory. Try and build the model from scratch but consult this pre-built model for guidance if needed.

Start with the Euler Falling Ball model and save it in your personal source code subdirectory as `EulerFallingBallVisualization.xml` to avoid over-writing the good copy. Create new variables `x0` and `vx0` in the analytic variables table with initial values of 0. Navigate to the View and select the *Windows, containers, and drawing panels* tab on the interface palette and create a frame by selecting the frame icon on the palette and then clicking on the Simulation view in the tree of elements. A *frame* is a top-level user interface element with a title and a border which serves as a container for other elements. Right-click on the name of the newly created frame in the tree and note that this frame is tagged as the *Main window* of the simulation. The main window is special because a running simulation will exit (terminate) when its main window is closed. The first frame created is assumed to be the main window, although you can change the main window designation by setting this property in another frame.

A frame is useless without content and we now create a *plotting panel* with x- and y-coordinate axes by selecting the Plotting panel icon on the palette. When the magic wand cursor appears, click on the plotting panel in the tree of elements to add a panel element to the frame by clicking on the frame in the tree. Two dialogs appear asking for the name of the plotting panel and the position of the panel within the frame. Choose the center position so that the plot occupies this region of the frame. Press the escape key or click in the tree of element white space to remove the wand. Otherwise you will create additional elements when you click on the tree. Double-click (or right-click) on the name of the newly created plotting panel in the tree to show its properties inspector. Set the Title property to `Falling ball` and the Title X and Title Y properties to `x` and `y`, respectively. *EJS* will add quotes to these properties because the charters typed are interpreted as a sequence known as a String. Set the Autoscale X property to `false` and enter `-5` and `5` for the Minimum X and Maximum X properties, respectively. Set the Autoscale Y property to `true` and enter `0` and `10` for the Minimum Y and Maximum Y properties, respectively. The y-axis will autoscale if objects lie outside these extremum but the x-axis will not autoscale. Set the Square property to `true` to insure that number of screen pixels per spatial unit is the same along both axes. Otherwise, geometric shapes may be distorted.

EJS has made a number of (reasonable) assumptions about what we have typed as we entered characters into the drawing panel inspector. *EJS* adds quotation characters to title properties because it assumes that titles are literal and should not be interpreted as variables. Behind the scene, *EJS* converts each title property into a Java `String` variable and uses it to set a (hidden) variable. The autoscale entries `true` and `false` are interpreted as Java `boolean` variables. Again, conversion is done behind the scene, and the boolean value is used to set the behavior of the plot. The extremum property values are interpreted as constants.

A plotting panel has titles and axes but it does not collect data or draw anything. We now add a ball and later create another panel to store and plot data. Navigate to the 2D drawables palette and select the shape icon from the collection of basic 2D elements. Add the shape to the plotting panel and use its inspector to set its X and Y position properties to `x0` and `y` and its size properties to 1.0. Set the shape's Draggable property to `true`. Select the arrow icon , add it to the plotting panel and use its inspector to set X and Y position properties to `x0` and `y`. Set the arrow's X and Y size properties to $vx0$ and vy, respectively, and set the X and Y scale properties to 0.25. Set the arrow's draggable and measured properties to `false` and its resizable property to `true`. Set the On Drag action property to

```
vx0=0;
```

Press, drag, and release actions occur in response to mouse actions when the simulation is running. Because we are building a one-dimensional model we always set the x-component of the velocity to zero after a mouse drag thereby insuring that the arrow only points in the vertical direction.

Exercise 2.7. Viewing the motion

Run the Euler Falling Ball Visualization model note that the ball in the view follows the position evolution the arrow follows the velocity of the model.

Make the following changes to the model so that the visualization runs smoothly:

a. Remove the print-line statements.

b. Set the time step to 0.01 so that the ball moves slower.

c. Set the FPS property on the evolution page to 10 so that the simulation computes ten frames per second.

Run the model again and observed the effect of these changes. ■

EJS's great strength is its ability to bind (connect) a model to its view without programming. The evolution algorithm changes the dynamical variables and the view is automatically updated. Furthermore, this binding occurs in both directions. Users can reposition the ball and change its velocity because we have set draggable and resizable properties to true. It is, however, difficult to interact with the ball as it is moving and we now add buttons to control the simulation.

A *panel* is a simple (lightweight) container without decoration, such as a title, that is used to group multiple user interface elements. Select the panel icon on the interface palette and click on the frame to create the element. Name this panel `controlPanel` and place it in the down position of the frame. Because we want to group our user interface elements near the bottom of the frame, create the panel in the frame's down position. Double click on the newly created panel's name in the tree of elements and set the Border Type property to `LOWERED_ETCHED`. You can type this value into the property field, but it is much easier to click on the editor icon (the first icon) to the right of the variable field to make the assignment. Note that the border name appears if you hover the mouse over an image in the editor. The control panel is almost invisible in the view because the panel has no content and its default size is zero except for the thin border.

Panels are often nested. Create a second panel named `buttonPanel` and place it within the control panel in the left position. Double click on this panel in the tree and change the Layout property to Grid Layout by clicking on the editor icon to the right of the property field. Set the grid's Rows and Columns properties to 1 and 4, respectively. Java containers, such as frames and panels, use a *layout manager* to position and size graphical user interface elements and the most commonly used managers are Border Layout, Flow Layout, and Grid Layout. The Border Layout is the default layout manager for frames and panels and the available positions for elements with this layout are `Center`, `Up`, `Down`, `Left`, and `Right`. We have chosen a grid because we wish to place four equally sized buttons in a rectangular array.

Navigate to the interface palette and select the two state button icon from the collection of buttons and decorations. Create a two state button in the button panel named `startStopButton`. Select the single state the button icon and create three additional buttons named `stepButton`, `resetTimeButton`, and `resetButton`. The two state button is frequently used to start and stop a simulation and its action properties are already set to invoke the appropriate methods. You should, however, use the inspector to set the button's text or to remove the text and set an image icon. The three remaining buttons do not have default actions.

Before we assign our button actions we create a *custom method* that resets the simulation time. Navigate to the Custom workpanel and create a new code page named `Reset Time`. Enter the following code in this page:

Listing 2.3: A custom method that resets the initial conditions.

```
public void resetTime () {        // begin method body
```

```
  t=0;     // initial time
  y=y0;    // initial position
  vy=vy0;  // initial velocity
}                               // end method body
```

The method declaration in line one consists of three parts: (1) The method is declared to be `public` so that it can be used anywhere within the program. (2) The keyword `void` tells us that the method will not return a result. (3) The name of the method `resetTime` tells us how the method is invoked. The method body follow the declaration in enclosed in curly braces and consists of three statements. We could have entered the body of the method in Listing 2.3 explicitly as a button action. However, defining a `resetTime` method promotes a cleaner and more readable model and allows us to give an example of a custom method to perform an action.

A button *action* is a block of code that is invoked when the button is pressed. Use the element inspectors to set the Action properties of the four buttons as shown in in Table 2.1. Each action is a single statement that invokes a method (function). The reset time button action invokes the custom method defined in Listing 2.3. The other buttons invoke predefined *EJS* methods. Because internal *EJS* methods and variables begin with the underscore character, users should not define methods or variables that begin with this special character. A complete list of predefined methods is available in Appendix 2A and in the online *EJS* WiKi by clicking on the information icon on the toolbar.

Table 2.1: Button actions in the Euler Falling Ball model visualization model.

Start on:	`_play()`	Plays (starts) the simulation.
Start off:	`_pause()`	Pauses (stops) the simulation.
Step:	`_step()`	Performs one evolution step.
Reset time:	`resetTime()`	Custom method that resets time $t = 0$.
Reset:	`_reset()`	Resets the simulation to its initial state.

We now add number fields to display the time and edit the time step. Create a new panel named `inputPanel` and place it within the control panel in the right position. Inspect this panel and change the Layout property horizontal box (`HBOX`). Add two additional panels to the input panel named `tPanel` and `dtPanel`. Each of these panels will display a label and a number field. Select the label icon **A** and create labels in the new panels in the left postion. Inspect these labels and set their text property to " t = " and " dt = ". Select the number field icon and create fields in the each panel's center postion. Inspect these number fields and set their Columns property to 4 and their Variable property to t and dt, respectively. Set the editable property for the time number field to false.

Exercise 2.8. Binding variables

Run the Euler Falling Ball Visualization model and note how the buttons behave. Note also that you can interact with the model by dragging the ball, dragging the tip of the arrow, and editing the time step. *EJS* binds (connects) the user interface to the model without programming. Add a number field to the user interface to set the initial velocity $v_y(t = 0)$. In addition to setting the initial velocity value, you should reset the time to zero. ■

Exercise 2.9. Polishing the model

Run and examine the pre-defined Euler Falling Ball Visualization model in the Chapter 2 source code directory. Are there differences between this model and the model you have constructed? Are the button sizes the same? How are button icons used in the predefined model? How is the Greek character delta Δ specified? ■

A consistent and attractive user interface conveys information clearly using standard discipline-specific terminology and notation. It takes little time to polish a model using *EJS* inspectors. Titles and labels can be changed and can incorporate special symbols. Borders and colors can be adjusted to group and highlight parameters. Element and variable names should reflect their function in the model. You will often start with default element names when building a model but you should right-click within the tree of elements and to change the default names before a model is distributed to others.

Exercise 2.10. Layout managers

It takes practice to learn how layout managers behave. Use the property inspector to change the control panel's layout manager to a flow layout. Resize the frame and observe how the user interface changes its appearance. Try a grid layout. Do either of these layouts make the user interface more attractive? Easier to use? ■

At this point, we hope you have gained a feeling for how *EJS* and Java work together. We have introduced typical *EJS* elements and have studied their inspectors. *EJS* inspectors are powerful tools for designing models because they provide two-way communication between the user interface and the model's global variables. Furthermore, inspector properties make reasonable assumptions about default values and most values need not be set. It is easy to see what element properties are important for the implementation of a model because those property values have inspector entries. Although unused property values are usually uninteresting, you can always right click within an empty input field to examine their default value.

Problem 2.2. Series user interface

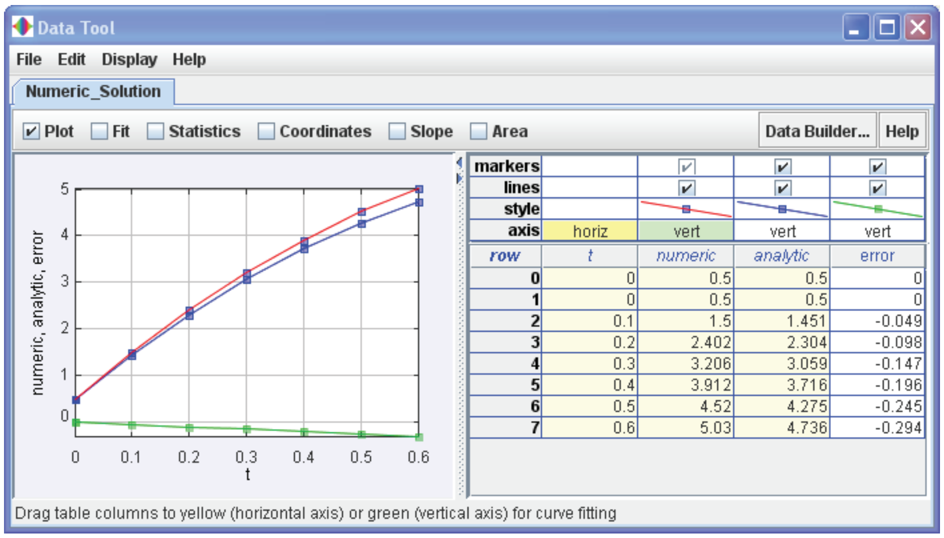
Design a user interface for the series sum described in Exercise 2.5. ■

Problem 2.3. Two-dimensional motion

Extend the Euler Falling Ball Visualization model to two dimensional motion. ■

2.4 Analyzing free fall

Time series data is generated while the Euler Falling Ball Visualization model is evolving but this data is lost because the model stores only the current y, v_y, and t values. Adding a plot to the model allows us to both visualize and analyze the data that is generated. Start with the Euler Falling Ball Visualization model, create a new frame named `dataFrame` containing a plotting panel named `dataPlottingPanel`, and save the modified model as `EulerFallingBallAnalysis.xml`. Navigate to the 2D drawables palette and select the Trace icon from the collection of basic elements and add a trail to the plotting panel named `yTrail`. Inspect the trail and set the Input X and Input Y properties to time t and position y, respectively. Set the X and Y table column labels to `time` and



markers		✓	✓	✓
lines		✓	✓	✓
style				
axis	horiz	vert	vert	vert

row	t	numeric	analytic	error
0	0	0.5	0.5	0
1	0	0.5	0.5	0
2	0.1	1.5	1.451	-0.049
3	0.2	2.402	2.304	-0.098
4	0.3	3.206	3.059	-0.147
5	0.4	3.912	3.716	-0.196
6	0.5	4.52	4.275	-0.245
7	0.6	5.03	4.736	-0.294

Drag table columns to yellow (horizontal axis) or green (vertical axis) for curve fitting

Figure 2.3: The Data Tool allows users to view, manipulate, and analyze data after it has been generated within *EJS*. Data Builder allows users to create new columns in the tool's table using functions of data in other columns. Fit Builder adjusts parameters with unknown values to fit data columns to functions.

`numeric`. Set the trail's line color to red. Add a second trail named `yAnayticTrail` to show the analytic solution. Set the X and Y table column labels to `time` and `analytic` and the line color to blue. We now test the accuracy of our model.

Exercise 2.11. Accuracy of the model

Run the Euler Falling Ball Analysis model and measure the maximum height of the ball by click-dragging within the graph. How accurate is your measurement? Does the model agree with a pencil and paper calculation of the maximum height? How long does it take for a difference to appear between the numerical and analytic solutions with a time step $\Delta t = 0.1$? With a time step $\Delta t = 0.01$? With a time step $\Delta t = 0.001$? ■

A graph reveals much about the behavior of a model but the precision of on-screen measurements is limited. There are two approaches to doing better data analysis. We can program the analysis ourselves or we can use a built in analysis tool.

Exercise 2.12. Numerical error

Add a position error plot to the Euler Falling Ball Analysis model. The error ϵ between the analytic y_a and numerical y_n position values can be computed as an absolute error

$$\epsilon = y_a - y_n \tag{2.11}$$

or as a relative error

$$\epsilon = (y_a - y_n)/y_a. \tag{2.12}$$

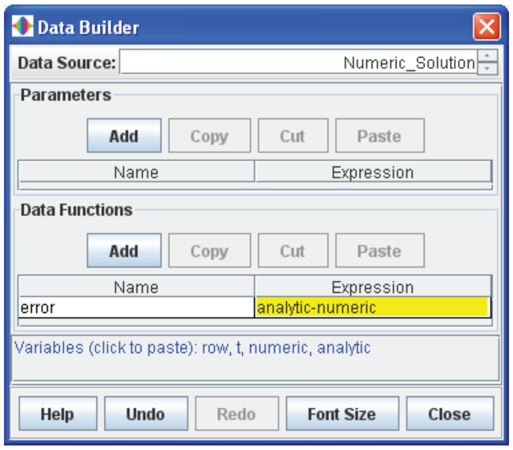
Which of these error calculations is most useful? How does changing the step size affect the absolute error plot? ■

Because analysis is an important aspect of modeling, *EJS* provides a general purpose data analysis tool that allow us to view, manipulate, and analyze data. This tool can be accessed in two ways. Run the model and evolve time to generate a graph. Right-click within the graph and choose [Element options] → [dataPlottingPanel] → [Data tool] from the popup menu. The Data Tool shown in Figure 2.3 appears. Note that you can change the name that appears within the popup menu by editing the Menu Entry property in the plotting panel's inspector.

Although it is straightforward to access the Data Tool using the popup menu, it is easier to access and control it using a button action method. Add a control panel to the bottom of the data plotting frame and add a button named `toolButton` to this panel. Feel free to change properties to improve the user interface appearance. Set the tool button action to:

```
_tools.showDataTool(_view.yTrail,_view.yAnalyticTrail);
```

This statement demonstrates an important and powerful programming concept. Names, such as `yTrail` and `yAnalyticTrail`, that are displayed in the tree of elements are variables and we can use these names to programmatically identify and to access them.[4] The elements in our model store data which we wish to analyze. The show tool statement uses the Java "dot" operator to identify the elements within the view and send these elements to the Data Tool for display.



Name	Expression
error	analytic-numeric

Variables (click to paste): row, t, numeric, analytic

Figure 2.4: The Data Build allows us to create new Data Tool columns using functions of existing columns.

Problem 2.4. Data Tool

Run the model with a time step $\Delta t = 0.1$ to generate time-series data and click the newly added tool button to show the Data Tool. The table columns display the numeric and analytic solutions.

[4]View elements are objects because they contain multiple parts just like String variables. This terminology seems natural. A car is an object that is made up of many parts but we refer to the entire car by a name. If we wish to refer to a part within the car we must specify which car and which part. We could, for example, refer to the left wheel on my car as `myCar.leftWheel`.

a. Use the Data Builder shown in Figure 2.4 to create a new column in the Data Tool. Add a data function named `error` and click within the function's expression field to set the new column's data to `analytic-numeric`.

b. The Data Tool uses least square fitting algorithms to fit the horizontal data column and the first vertical data column to a function with unknown parameters. Select the *fit* checkbox, choose a parabolic fit, and select autofit. Note that you can drag data columns to change their table position. How accurately does the numerical solution fit the quadratic coefficient? The linear coefficient?

c. Drag the column with the computed error to the first position and fit the data. What polynomial fitting coefficient changes when the time step Δt is decreased and how does this fitting parameter vary with Δt?

Note that you can change the graphical display by clicking on the style field above the data columns. You can also right click on the tab to create additional data tabs and delete unnecessary columns so as to simplify the Data Tool display. ■

Java has many classes and dot notation allows programmers to organize mountains of code using a tree-like hierarchy. For example, the Math class contains methods for performing basic numeric operations such as the elementary exponential, square root, and trigonometric functions. Math class methods (functions) are said to be *static* because the algorithms do not change and the result depends only on the arguments. Static methods are invoked using the name of the class followed by the method name. For example, in order to calculate the sin of 0.5 radians we write `Math.sin(0.5)`.

View elements can also be accessed using dot notation but they must be identified using the name that is displayed in the tree of elements. Each element is constructed from the same template but can have different property values. In other words, a model can contain multiple trails and each trail has its own xy-data and drawing properties such as color. We can, for example, add xy-point data to the model's `yTrail` using the statement

```
_view.yTrail.addPoint(t, y);
```

but this low level programming is seldom necessary because *EJS* adds data points automatically using variables that are bound (connected) to inspector input properties.

2.5 Multiple particles

We now create a model showing balls bouncing within a box as shown in Figure 2.5 . We build this new model from scratch because there is little overlap with previously constructed single ball models but we reuse the control panel from the Euler Falling Ball Visualization model. Open this old model, right-click on the control panel in the tree of elements, and select copy from the list of options. Click the create a new model button 🗋 on the *EJS* toolbar, create a new main frame, select the frame in the tree of elements, and paste the clipboard contents into the frame's down location. You can also save the clipboard contents as a *compound element* for use in future projects by right clicking on the first palette in the Interface palette group. Right-click on the reset time

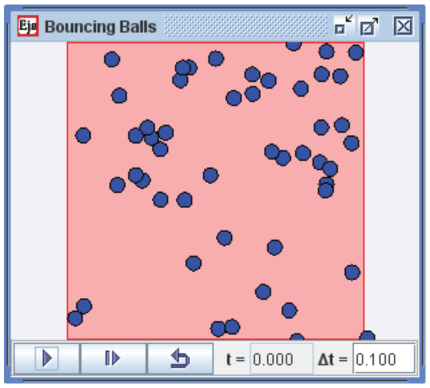


Figure 2.5: The Bouncing Ball model shows the motion of non-interacting balls bouncing within a 2D box.

button in the tree and cut (delete) it from the button panel. Create a drawing panel (not a plotting panel) in the center of the main panel.

Inspect the number fields in the control panel and note that the input field Variable properties have pink backgrounds because the time t and time step Δt variables have not yet been defined.[5] Navigate to the Variables panel and create `t` and `dt` variables and note the inspector field backgrounds have returned to white. Create an integer variable `n` with a value of 50. Be sure to change the data type for `n` from `double` to `int` and do not include a decimal point when entering the value. Integer data is stored and processed differently from real (floating point) numbers and the value field will signal an input error by turning pink if you use an incorrect format when entering a value. Create a double variable named `boxSize` and set its initial value to 5 in order to define a bounding box for the particles. Set the drawing panel's minimum and maximum scale property values to `-boxSize` and `boxSize`, respectively, and set the Square property to true.

Create variables `x`, `vx`, `y` and `vy` to store position and velocity. Set the data type to `double` and the dimension to `[n]` for these variables. A dimensioned variable is an array that holds many variables of the same type. The elements of an array are accessed using an index in square brackets. Because the index begins at 0 and ends at the length of the array n minus 1, the last element of the x-position array is accessed and set using

```
int n=x.length;   // arrays have a length property
x[n-1]=5.0;       // maximum element index is length-1
```

An array out of bounds error occurs if the index is less than zero or greater than $n - 1$ when the program is run. To make the model we are building interesting, we now use the position and velocity arrays to animate the motion of non-interacting 2D balls (circles) with random initial velocities.

Initial value column entries within a Variables Table set values using a constant or a simple expression. The Initialization workpanel gives us the opportunity to write additional code. Navigate

[5] *EJS* checks inspector values for errors. You can enable error checking using the options button on the toolbar or you can search for errors using the search button on the toolbar.

to the Initialization workpanel and enter the code in Listing 2.4.

Listing 2.4: Bouncing ball position and velocity initialization.

```
double v=2;  // initial speed
for(int i=0; i<n; i++){                      // beginning of loop
  x[i]=boxSize*(-1+2*Math.random());        // random x
  y[i]=boxSize*(-1+2*Math.random());        // random y
  double theta=2*Math.PI*Math.random();// random angle
  vx[i]=v*Math.sin(theta);                   // x velocity component
  vy[i]=v*Math.cos(theta);                   // y velocity component
}                                            // end of loop
```

This initialization code uses a loop to set each array element to a random value using a Math class method that returns a random number within the interval $[0, 1)$. Velocity components are set using a random angle and trigonometric functions.

We again emphasize that where we define variables is important. The `x`, `vx`, `y`, `vy`, `boxSize`, and `n` variables are defined in the model's variables table and can therefore be used throughout the model because they are global. The `v` and `theta` variables are defined in the initialization code page and can only be used within that page after they are defined. The counter variable `i` is defined in the `for` loop and can only be used within that loop. These variables are *local*. It is considered bad programming to clutter up a model with unnecessary global variables. It is far better to define and use local variables on the code page where they are needed. A short end of line comment is all that is needed to document a local variable's purpose. Note that if we define a local variable using a global variable's name, for example, if we had written `int n=10`, then the local variable hides (shadows) the global variable and the global variable becomes inaccessible.

Enter the code in Listing 2.5 into the Evolution workpanel to advance the positions of the particles.

Listing 2.5: The evolution moves particles with constant velocity within a bounding box.

```
for(int i=0; i<n; i++){ // creates loop counter i and starts loop
  x[i] += vx[i]*dt; // += operator adds RHS to LHS; store result in LHS
  y[i] += vy[i]*dt;
  // reverse velocity component to keep balls inside box
  if(x[i]>boxSize && vx[i]>0) vx[i] = -vx[i];
  if(x[i]<-boxSize && vx[i]<0) vx[i] = -vx[i];
  if(y[i]>boxSize && vy[i]>0) vy[i] = -vy[i];
  if(y[i]<-boxSize && vy[i]<0) vy[i] = -vy[i];
}  // end of loop
```

The algorithm uses a loop to apply Euler's method to each ball but reverses the velocity component if the ball is outside the box and moving away from a wall. The Java boolean *and* operator `&&` returns true if both operands are true. The Java boolean *or* operator `||` returns true if one operand is true.

Conservation principles, such as the conservation of energy, provide a useful check for the correctness of many physical models and we now compute the total mechanical energy to test our model. Create variables for energy and mass in the variables table and then enter the code in Listing 2.6 into the fixed relations workpanel.

Listing 2.6: The fixed relation code computes the total mechanical energy of the particles.

```
energy=0;                        // sum starts at zero
for(int i=0; i<n; i++){          // add particle energies
  energy += 0.5*m*(vx[i]*vx[i]+vy[i]*vy[i]);
}
```

Euler's method is exact if the particle velocity is constant and the energy should not drift if we have implemented our model correctly. Our model does have position errors because our particles move beyond the box boundary before their velocity component's are reversed and we will soon see energy errors as we add interactions (forces). Inspect the drawing panel and enter the expression

```
"Energy="+_format(energy,"0.0000")
```

into the drawing panel's TL Message property to display the energy. The `_format` method is a predefined *EJS* method that converts a number into a string using the given pattern.

We refer to variables that are computed using the evolution algorithm as *state variables* or *dynamical variables*, and we refer to variables that depend on these variables as *auxiliary variables* or *output variables*. Evolution pages are evaluated a certain number of times per second when the play button is pressed but fixed relations are evaluated after the model is initialized or reset and whenever the model changes. The fixed relation will, for example, be evaluated when a user drags a particle or changes an input parameter.

Listings 2.4, 2.5, and 2.6 are all that is required to describe the simple physics of non-interacting 2D particles confined to a box. We now add the visualization. Navigate to the *sets of 2D drawables* palette, create a 2D shape in the drawing panel, and name it `box`. Inspect the box and set its X and Y Size properties to `2*boxSize`. Set the Style property to `RECTANGLE`, the Line Color property to `RED`, and the Fill Color property to PINK. Create a *shape set* named `particles` in the drawing panel. A set of elements is a collection of elements whose properties can be set using arrays. Show the particle set's inspector and set the Position X property to `x`, the Position Y property to `y`, the Fill Color to `BLACK`, and the Draggable property to `true`. Sets of elements are easy to use because each element is bound (connected) to a corresponding values in the x and y arrays. The default # Elements property is set by the dimension of the x variable. If a property value, such as the Fill Color, is constant, then the set assigns the same property value to all elements in the set. Run the simulation and observe the balls moving and bouncing off the walls.

Problem 2.5. Falling balls

Add a constant downward acceleration (gravity) to the Bouncing Ball model. How large a value of acceleration can be used before the energy drift becomes apparant when the time step is $\Delta t = 0.1$? Reduce the time step to $\Delta t = 0.01$ but increase the steps per display SPD to 10 so that the speed appears to remain constant. How does the smaller time step affect the energy drift? ■

The paradigm of binding array variables to properties of element sets makes it easy to show the particle velocities without additional programming. Use the *EJS* WiKi to learn about arrow sets before doing Exercise 2.13.

Exercise 2.13. Balls with arrows

Add an arrow set to the display panel and set its position and size properties so that the arrows follow the balls and display the particle velocities. ■

2.6 Summary

We have described many of the Java and *EJS* concepts that we will need to implement our models. In particular, we have introduced frames, panels, and input output elements such as buttons and fields. We have introduced Java variables and arrays and have shown how they are used in Java statements. We have introduced Java objects and hinted at the concepts of *object oriented programming* that are the foundation of modern programming. We have also described many of the behind the scene operations that occur when we build a graphical user interface. Additional aspects of *EJS*, such as 3D modeling and the ordinary differential equation editor, will be taught by example but we will no longer describe the details of how to construct a user interface.

A word about simplicity of design. *EJS* makes it easy to create user interface elements for every parameter and every initial condition in a model. Resist the temptation to include every possible option and to provide input fields for every variable. Good models focus the user's attention on parameters or combinations of parameters that significantly affect the outcome. The best models are those that capture the relevant science with a small number of parameters. This Occam's Razor description of the *best model* is an uncomputable task. In the final analysis, the best model is the one that works best for you and the model's users.

Problem 2.6. Gravitational potential energy

Use *EJS* to plot the gravitational potential starting at the Sun's surface, passing through the center of the Earth, and continuing outward until the distance from the Sun is twice the radius of Earth orbit. (Assume the gravitational potential is zero at infinity.) Set the scale so that Earth's position is at the center of the plot and include a slider that zooms in on the plot center.

The completed model should contain a documentation page with a discussion of the significance (importance) of the Earth's gravitational field for objects in orbit about the Sun. Describe the programming difficulties that needed to be solved to create the plot and describe the computation of the gravitational potential when passing through the Earth. ■

2.7 Simulations

The following models are implemented in *EJS* and are downloadable from the OSP Collection in the ComPADRE digital library.

Euler Method

The Euler Method model introduces students to first-order differential equations using a ready-to-run implementation of Euler's algorithm. The user enters a differential equation $dy/dx = f(x)$, the initial value y_0, and the step size Δt. The simulation displays the solution in a table with columns showing the solution step counter n, the independent variable x, and the solution y. The numerical solution y is an approximation of the true solution because of approximations inherent in Euler's algorithm. See Section 2.1.

Euler Fall Ball

The Euler Falling Ball model is a simple example designed to teach Java syntax. The simulation does not have a user interface and prints its output in the EJS console. Although the model has only eleven lines of computer code, EJS builds a complete Java program, compiles the program, and runs the program when the the taskbar's run button is pressed. This model shows (1) how to write Java statements and using various types of variables, (2) that numerical errors accumulate rapidly when using Euler's method, and (3) that computer arithmetic is not exact.See Section 2.2.

Euler Fall Ball Visualization

The Euler Falling Ball Visualization model adds a graphical user interface to the Euler Falling Ball model. This Euler Falling Ball Visualization model shows (1) how to display moving objects how to display data, (2) how to start, stop, and reset a simulation, and (3) how to input data to a simulation. See Section 2.3.

Euler Fall Ball Analysis

The Euler Falling Ball Analysis model introduces the Data Tool to analyze Euler's numerical method solution of the fall particle model. See Section 2.4.

Bouncing Balls

The Bouncing Ball model shows the motion of fifty non-interacting balls bouncing within a 2D box. This model introduces array syntax and shows how arrays are used to position objects within the view. The position arrays `x[50]` and `x[50]` are connected (bound) to the X- and Y-Position properties of a Shape Set and the elements of this set automatically adjust their location as the values of the array elements evolve within the model. Because each array elements is bound to its respective shape, users can drag the shape to set array values. See Section 2.5.

Appendix 2A: *EJS* Reference

Predefined Methods

EJS predefines utility methods to control the model's execution and to perform other common operations. There are methods that stop and start the simulation, that return the simulation to its initial state, and that store and read the simulation's state to and from a file as described in the following tables. These methods are easy to access by right-clicking within a code page and selecting the appropriate category as shown in Figure 2.6. All predefined methods and internal variables begin with the underscore _ character.

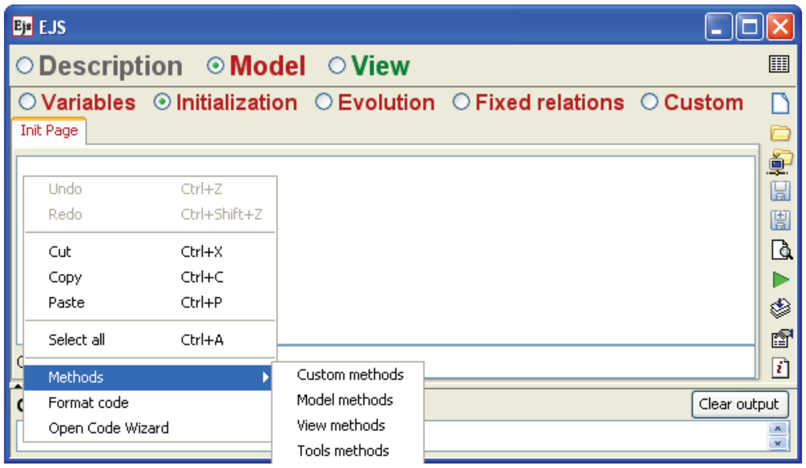


Figure 2.6: A popup menu is used to access lists of methods from within a code page.

Model methods to control the execution of the simulation.
`int _getDelay()` Returns the current delay time in milliseconds. The delay time is the reciprocal of the frames per second (FPS) property.
`void _initialize()` Executes the Initialization workpanel code without resetting model variables to their default value. This method is often used to re-initialize a simulation with user interface values because it does not reset global variables to the values in the table of variables.
`boolean _isApplet()` Determines if the simulation is running in an applet (web browser) environment.
`boolean _isPaused()` Determines whether the evolution of the simulation is not playing. It is the opposite of the `_isPlaying()` method.
`boolean _isPlaying()` Determines if the evolution is playing. This method is often used to enable or disable elements of the view.
`void _pause()` Pauses the evolution at the end of the current evolution step.
`void _play()` Starts (plays) the evolution. The evolution proceeds at a rate determined by the frames per second (FPS) property and the step per display (SPD) property. If FPS is 10 and SPD is 5 the program will execute the evolution algorithm 50 times per second and will collect data at the end of every step but will only redraw the screen 10 times per second.
Continued on next page.

Model execution methods from previous page.
`void _reset()` Resets the simulation to its initial state. This method clears all data, sets global variables to the initial value in the table of variables, and sets all user interface elements to their default values.
`void _setDelay(int msec)` Sets the delay time in milliseconds between model's evolution steps. The delay time is the reciprocal of the frames per second (FPS) property.
`void _setFPS(int fps)` Sets the number of frames per second (FPS) of the simulation to the given value. The default FPS parameter is displayed on the Evolution workpanel.
`void _setSPD(int steps)` Sets the number of steps per display. This number indicates how many evolution steps are computed before updating the view. View elements collect data after every step and the view is refreshed after the last step. Setting a high number of steps per display can speed up the simulation considerably because unnecessary screen redrawing is avoided. The SPD parameter is displayed in the Evolution workpanel and is set to one by default.
`void _step()` Performs SPD evolution steps and refreshes the display (see the `_view.setSPD()` method). Fixed relations are evaluated after every evaluation of the evolution so that multiple data points are generated and collected if SPD is greater than one. If the simulation is running, the evolution pauses at the end of the current step.

Model IO methods to save and read data.
`String[] _getArguments()` Returns the in command-line arguments when the simulation is run from a shell window.
`boolean _readState(String filename)` Reads the previously saved model state. This method reads the value of all the variables of the model from the file. If the filename ends with the suffix ".xml", the data is assumed to be a text file in XML form. Otherwise, the data file is considered to be binary and then caution is needed in the sense that the data file must have been created with the `_saveState` method from the same simulation with exactly the same variables and declared in the same order. If successful, the method returns a true boolean and false otherwise. If the name of this file begins with the keyword "URL", the method will read the rest of the name as an Internet address and will try to connect to this address. If successful, it will play the simulation from the data on the web server in question. If the rest of the name begins with "http:", then the Internet will be used as the source of the data file.
`String _readText(String filename)` Reads text from a given file. Returns null if there is any I/O error.
`boolean _readVariables(String filename, String variablesList)` Reads a list of variables from a file. The list of variables is a String with the variable names separated by commas, spaces, or semicolons.
Continued on next page.

Model IO methods continued from previous page.
`boolean _readVariables(String filename, java.util.List<String> variablesList)` Similar to the previous one, but now the list of variables is given as a java.util.List of Strings.
`boolean _saveImage(String filename, String elementName)` Saves an image of the given view element to file.
`boolean _saveState(String filename)` Saves the model state. This method saves the current values of global variables defined in the Variables tables in the specified file. It returns a true boolean if successful. This methods returns false if it encounters an IO error, such as a write protected output location. The file name specifies the relative or absolute path where the file will be created. If the filename has a ".xml" suffix, a text file in XML form will be created. In all other cases, the file will be created in binary form. Binary files can be considerably smaller than XML files, but the `readState` method will then read the variables blindly in the order in which they appear in the model. Hence, if you modify the number of variables or their order of declaration in a model, previously saved data may be read incorrectly. If the filename begins with the prefix "ejs:", then the data is stored only in memory. A model can read this data as if it were a disk file (using the method `readState`) during the current session, but that data will be lost at the end of the session. Temporary storage is useful for storing various intermediate stages during the same session. In particular, this is of interest when the application runs as an applet, since (for security reasons) an (unsigned) applet can not write to the hard disk.
`boolean _saveText(String filename, String text)` Saves text to file.
`boolean _saveVariables(String filename,String variablesList)` Saves only the prescribed list of variables to a file. If the filename ends with ".xml" the file is a text file in XML form. Otherwise, the file is binary and must be read with the `_readVariables` method with the same list of variables. The list of variables is a String with the variable names separated by commas, spaces, or semicolons.
`boolean _saveVariables(String filename, java.util.List<String> variablesList)` Similar to the previous one, but now the list of variables is given as a java.util.List of Strings.

Model methods to control *EJS* applets using Java Script.
`String _getParameter(String parameterName)` Returns the applet's parameter with this name. Null if the parameter is not specified in the applet.
`boolean _setVariables(String command, String sep, String arraySep)` Sets the value of one or more variables of the model. The command string must consist of pairs of the type 'variable=value' separated by the separator string `sep`. The arraySep string is used as the separator string for values of array variables. The method returns true if all the variables are correctly set.
Continued on next page.

Model applet methods continued from previous page.
`boolean _setVariables(String command)` Equivalent to `setVariables (command, ";", ",")`.
`String _getVariable(String variableName)` returns a String with the value of a public variable of the model. If the variable is an array, individual element values are separated by commas. Only public variables of primitive or String type can be accessed by this method.

EJS-created applets have also public fields called `_model`, `_simulation`, and `_view`, which provide access to all other predefined and custom methods. A method you may need to use frequently in applet mode is the `_simulation.update()` method. This method is called automatically whenever the user clicks on a button in the view. However, clicking on JavaScript controls do not update the simulation and requires you to invoke this method explicitly.

Tool methods.
`_tools.clearDataTool()` Clears all data from the data tool.
`_tools.clearFourierTool()` Clears all data from the Fourier analysis tool.
`Object _tools.showDataTool(Data... _view.dataElement)` Same as `_tools.showDataTool(null, Data... _view.dataElement)`.
`Object _tools.showDataTool(Component _view.parentComponent, Data... _view.dataElement)` Similar to the previous method, but here all data is considered to belong to a same series and appears in a single tool tab with the first column considered to be equal.
`Object _tools.showDataTool(Component _view.parentComponent, java.util.List<Data> _view.dataElement)` Displays data of one or several elements. Each data in the list is considered to be independent and appears in a separate tool tab. If this method is used repeatedly with the same data, the data in the data tool is updated. The data tool appears relative to the given view element or centered if null. Returns the data tool.
`Object _tools.showFourierTool(Data... _view.dataElement)` Similar to the matching showDataTool method but using the Fourier analysis tool.
`Object _tools.showFourierTool(Component _view.parentComponent, Data... _view.dataElement)` Similar to the matching showDataTool method but using the Fourier analysis tool.
`Object _tools.showFourierTool(Component _view.parentComponent, java.util.List<Data> _view.dataElement)` Similar to the matching showDataTool method but using the Fourier analysis tool.
`Object _tools.showTable(Data... _view.dataElement)` Same as `_tools.showTable(null, Data... _view.dataElement)`.
Continued on next page.

Tool methods continued from previous page.

`Object _tools.showTable(Component _view.parentComponent,`
`                        Data... _view.dataElement)`
Creates a table with the data of one or several elements. The table appears relative to the given view element or centered if null. Returns the ArrayFrame object which contains the table.

View methods.
`void _view.clearData()` Clears data from the elements that store them (such as trails, traces, or tables).
`void _view.clearElements()` Clears data from the view. The effect of this method depends on the type of element. Unlike the `_view.resetElements()` method, this method should not affect formatting. See also `_view.clearData()`.
`Element _view.getElement(String elementName)` Returns the *EJS* element with the given name. This element is a wrapper object for the underlying Java object, which can be obtained using the construction `_view.elementName`. The wrapper object is seldom used, but can be used by advanced users to change the element properties in run-time.
`java.awt.Container _view.getTopLevelAncestor(String elementname)` Returns the top-level ancestor of the given view element (either the containing Window or Applet), or null if the element has not been added to any container. If no element name is provided, the first view element whose component is a Window is returned.
`void _view.resetElements()` Resets view elements to their original state. The effect depends on each type of view element. For example, this method removes stored data from trace and trail elements and removes formatting and other programmatic customization. See also `_view.resetTraces()`.
`void _view.resetTraces()` Same as `_view.clearData()`.
`void _view.setUpdateView(boolean update)` Indicates whether to update the view every time after an evolution step. Default is true, but disabling the updating of the view (i.e. invoking this method with a false parameter) can be useful to have the model run as fast as possible, not losing any CPU effort in refreshing the view. When the computation is done, the program can use this method again with a true parameter.
`void _view.update()` Updates all view elements with the current values of the model's global variables.

View methods to display documents.
`JDialog _view.createDescriptionDialog(Component owner, String pageName)` Returns a javax.swing.JDialog displaying the named description page and the specified window as owner. The owner window may be null.
`JDialog _view.createDialog(Component owner, Component child)` Returns a javax.swing.JDialog with the given owner and child component
`JDialog _view.createHTMLDialog(Component owner, String pageURL)` Returns a javax.swing.JDialog displaying the given HTML page and the specified window as owner. The owner window may be null.
`JDialog _view.createHTMLDialog(Component owner, java.net.URL aURL)` Returns a javax.swing.JDialog displaying the given HTML page and the specified window as owner. The owner window may be null.
`JScrollPane _view.createHTMLPage(String pageURL)` Returns a javax.swing.JScrollPane displaying the given HTML page.
`JScrollPane _view.createHTMLPage(java.net.URL url)` Returns a javax.swing.JScrollPane displaying the given HTML page.
`java.net.URL _view.getDescriptionPageURL (String pageName)` Returns the URL of the HTML page corresponding to the description page with the given name. This URL locates the description file inside the JAR file of the simulation and can be used programmatically to display a single page of the description.
`void _view.showDescription(boolean show)` Shows/hides the description dialog that is distributed in the simulation's jar file.
`void _view.showDescriptionAtStartUp(boolean show)` Indicates whether the description of the simulation must be displayed at start-up. Default is true, but users can disable it in order to display programmatically the description, or individual pages of it.
`_view.showDocument(String document)` Opens the given document using the native application, if a document reader is defined.

View methods to display messages.
`void _view.alert(String elementName,String title, String message)` Displays a message in an alert dialog window with the given title and message. The simulation pauses until the dialog is closed. This method is useful for displaying error messages or warnings that the user should not ignore. The dialog is centered on the given view element or centered on the screen (or on the last parent component specified by the `setParentComponent` method) if a null element name is given.
`void _view.clearMessages()` Clears all text from the last created text area of the view (if any). See `_view.println()`.
`String _view.format(double value, String template)` Converts a double value to a string with the given template. Example: `_format(Math.PI, "0.00"` print the number PI with two decimal digits.
Continued on next page.

View methods to display messages continued from previous page.
`void _view.print(String message)` Prints the given message in the last created text area of the view. If the view has no text area elements, the message will appear in the standard output.
`void _view.println(String message)` Prints the message followed by a new line character.
`void _view.println()` Prints an empty new line.
`void _view.setParentComponent(String elementName)` Sets the parent element for displaying dialog windows, such as alerts.

Appendix 2B: Java Syntax

Operators

Operators such as `++` and `+=` are are convenient shortcuts that improve the efficient of your computer code. Table 2.9 shows the operators that we will use most often.

operator	operand	description	sample expression	result
`++, --`	number	increment, decrement	`x++;`	8.0 stored in `x`
`+, -`	numbers	addition, subtraction	`3.5 + x`	11.5
`!`	boolean	logical complement	`!(x == y)`	true
`=`	any	assignment	y = 3;	`3.0` stored in `y`
`*, /, %`	numbers	multiplication, division, modulus	`7/2`	3.0
`==`	any	test for equality	`x == y`	false
`+=`	numbers	`x += 3`; equivalent to `x = x + 3`;	`x += 3;`	14.5 stored in `x`
`-=`	numbers	`x -= 2`; equivalent to `x = x - 2`;	`x -= 2.3;`	12.2 stored in `x`
`*=`	numbers	`x *= 4`; equivalent to `x` = 4*x;	`x *= 4;`	48.8 stored in `x`
`/=`	numbers	`x /= 2`; equivalent to `x = x/2`;	`x /= 2;`	24.4 stored in `x`
`%=`	numbers	`x %= 5`; equivalent to `x = x % 5`;	`x %= 5;`	4.4 stored in `x`

Table 2.9: Common operators. The result for each row assumes that the statements from previous rows have been executed with `double x = 7, y = 3` declared initially. The *mod* or *modulus* operator, `%`, computes the remainder after the division by an integer has been performed.

The Math class

Standard mathematical functions are defined in the Math class that is distributed with Java. To use these methods you must use the prefix "Math." For example, in order to calculate the sin of 0.5 radians, you must write: `Math.sin(0.5)`. Methods in the Math class are static because the algorithms do not change and the result depends only on the argument.

Methods from the Math Library in Java
`double abs(double x)` Returns the absolute value of double value **x**.
`double acos(double x)` Returns the arc cosine of **x**, the returned angle is in the range of 0 to π.
`double asin(double x)` Returns the arc sine of **x**, the returned angle is in the range of -$\pi/2$ to $\pi/2$.
`double atan(double x)` Returns the arc tangent of **x**, the returned angle is in the range of -$\pi/2$ to $\pi/2$.
`double ceil(double x)` Returns the smallest double value integer that is greater than or equal to **x**.
`double cos(double x)` Returns the cosine of the angle **x**.
`double exp(double x)` Returns Euler's number e to the power of the double value (**x**), e^x.
`double floor(double x)` Returns the largest double value integer that is greater than or equal to **x**.
`double log(double x)` Returns the `natural logarithm` (base e) of the double value **x**.
`double max(double x, double y)` Returns the greater value of the two double values **x** and **y**.
`int max(int x, int y)` Returns the greater value of the two integer values of **x** and **y**.
`double min(double x, double y)` Returns the lower value of the two double values of **x** and **y**.
`int min(int x, int y)` Returns the lowest value of the two integer values of **x** and **y**.
`double pow(double x, double y)` Returns the value of the first argument (**x**) raised to the value of the second argument (**y**), (e^{ylogx}).
`double random()` Returns a randomly selected positive double value, this number is between 0.0 and 1.0 (excluding 0.0).
`double rint(double x)` Returns the closest double value integer that to argument **x**.
`long round(double x)` Returns the closest **long** to argument **x**.
`double sin(double x)` Returns the sine of angle **x**.
`double sqrt(double x)` Returns the correctly rounded square root of double value **x**.
`double tan(double x)` Returns the tangent of angle **x**.
Continued on next page

Continued from previous page
`double atan2(double x, double y)` Returns the angle *theta* from the conversion of rectangular coordinates (**x**,**y**) to polar coordinates (*r*,*theta*).

References and Suggestions for Further Reading

By using the *EJS* we have hidden most of the Java code needed to create a Java program. There are many good books on Java and we list a few of our favorites in the following. The online Java documentation provided by Sun at `<java.sun.com/docs/>` is essential (look for API specifications), and the tutorial, `<java.sun.com/docs/books/tutorial/>`, is very helpful. There are many other useful tutorials on the Web.

Bruce Eckel, *Thinking in Java*, fourth edition, O'Reilly (2006).

Cay S. Horstmann and Gary Cornell, *Core Java, Volume 1 – Fundamentals*, eight edition, O'Reilly (2007).

Pat Niemeyer and Jonathan Knudsen, *Learning Java*, third edition, O'Reilly (2005).