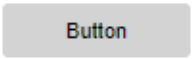
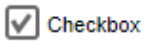
Chapter 13. Designing interactive models: using controls

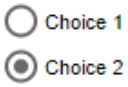
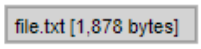
You can make your AnyLogic models interactive by including various *controls* (buttons, sliders, text inputs, etc.) into the model front end, and also by defining reactions to mouse clicks. The controls can be used both to set up parameters prior to the model execution and to change the model on-the-fly.

Controls can be found in the **Controls** palette and are created and edited in just the same way as shapes. Controls can be grouped with shapes and other controls and can be replicated. Just like shapes, controls have dynamic properties that can be used to change their size, position, availability, and visibility at runtime. Controls of the agent can be set to appear on the container agent presentation.

Controls always appear on top of any other graphics (shapes, model elements, etc.) regardless of the Z-order and grouping. You should avoid overlapping controls as this may produce undesirable visual effects.

Controls that have state or content (such as slider, radio buttons, edit box, etc.) in AnyLogic have *value* and can be *linked* to variables and parameters, so that when the user changes the control state, the linked element changes too (but not vice versa). In addition, you can associate an arbitrary action with a control, e.g. call a function, schedule event, send message, stop the model, and so on. The action gets executed each time the user touches the control. The value of the control is typically available as **value** in the control's **Action** code field and also is returned by the control's **getValue()** function. Quick information about each control is given in the Table below.

Control	Type of value	Can be linked to type	Comments
Button 			Is used to perform custom immediate actions in the model. You write code in the **Action** field and that code gets executed when the user clicks the button.
Checkbox 	**boolean**	**Boolean**	

Edit box 	**String**	**String** or any numeric type (**double**, **int**, etc.)	In addition to linking the edit box to a variable, you may define your own custom code that handles (validates, accepts, rejects) the user's input.
Radio buttons 	**int**	**int**	The first choice corresponds to value 0, the second – to 1, and so on.
Slider 	**double**	**double** or any numeric type (**int**, etc.)	You can define the minimum and maximum values of the slider.
Combo box 	**String**	**String**	Can be editable or fixed which limits choices to a defined set.
List box 	**String**	**String**	Can work in single or multiple selection modes. If you choose **Multiple selection**, the list box cannot be linked and its value is available via the **getValues()** function that returns the array **String[]**.
File chooser 	**String**		In the **Upload** mode, displays the system **Open** dialog prompting the user to select the file. In the **Download** mode, downloads the specified file to the default download location. In both modes keeps the result as **String** containing the file name with full path.
Progress bar 	**double**	**double** or any numeric type	In the deterministic mode, displays a given progress value. In the nondeterministic mode, displays "activity is going on". Both **Progress value** and **Determinated** properties are dynamic, i.e. constantly evaluated at runtime.

Please note that **List box, File chooser** and **Progress bar** are available in AnyLogic Professional and AnyLogic University Researcher editions only.

Example 13.1: Slider linked to a model parameter

We will create a parameter and a slider and link them. We will continue this example later and build upon it.

Follow these steps:

1. Create a parameter by dragging the **Parameter** element from the **Agent** palette to the graphical editor.
2. In the properties of the parameter set its default value to **50**.
3. Open the **Controls** palette and drag the **Slider** element. Place it next to the parameter. Extend the slider a bit as shown in Figure 13.2.
4. In the properties of the slider check the **Link to** checkbox and select **parameter** from the drop-down list on the right.
5. Set the following properties:
   Minimum value: **0**
   Maximum value: **100**
6. Run the model. Move the knob of the slider and watch the parameter value.

As you may have noticed, the slider position is set to the initial value of the parameter (50) when the model starts. If the value of the parameter is outside the range of the slider, the slider will be moved to the closest possible position, but the value of the parameter will not change until you touch the slider.

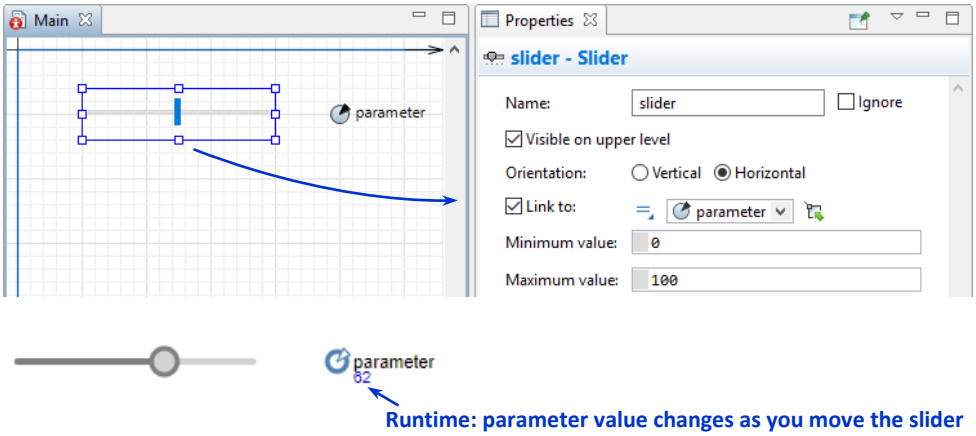


Figure 13.2 Slider linked to a parameter

You should keep in mind that if the value of the parameter is changed "externally", i.e. not by the slider, the slider position will *not* be automatically adjusted. If you

wish the link to always work "both ways" you can, for example, add the corresponding code in the **On change** field of the parameter (see the next example).

Example 13.2: Buttons changing the parameter value

We will add two buttons to the previous example. The buttons will increase and decrease the parameter value by 1, yet keeping the parameter in the range 0-100.

Add the buttons:

1. Create two buttons (use the **Button** element from the **Controls** palette) and arrange them as shown in Figure 13.3.
2. In the properties of the left button set its **Label** to **-1**.
3. Click the **Enabled** property's icon to switch to the dynamic value editor. Type **parameter >= 1** in the **Enabled** field.
4. Expand the **Action** section of the button properties and type **set_parameter(parameter-1);** in the code field.
5. Select the right button and adjust its properties in the same way:
   Label: **+1**
   Enabled: **parameter <= 99**
   Action: **set_parameter(parameter+1);**
6. Run the model. Move the slider and use buttons.

Do not confuse the *name* of a control with its *label*. The name is the name of the Java object created for the control (and used to access the control's API), while the label is the text that appears near or on the control on the screen. The label can be changed dynamically during runtime.

The (auto-generated) function **set_parameter(parameter-1);** is called in the button's action instead of simpler code **parameter--;** to make sure the parameter is changed correctly, in particular its **On change** code is called. The expression you enter in the **Enabled** field of a control is constantly evaluated during runtime, and, if it evaluates to **false**, the control gets disabled, and vice versa. In our case we do not want the user to be able to drive the parameter value out of range 0-100, so when the value is closer than 1 unit from the range border, we disable the corresponding buttons.

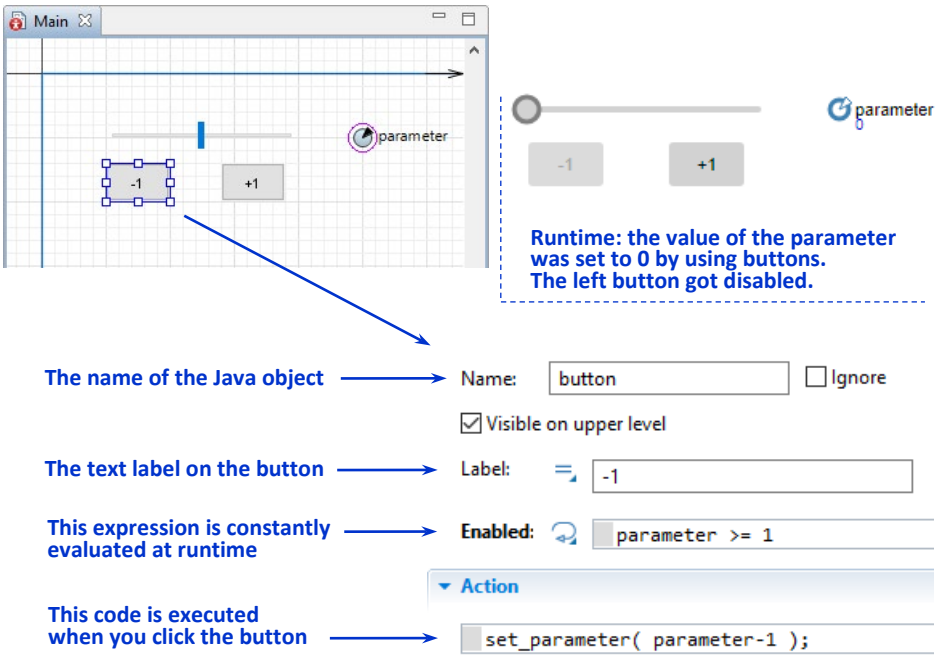


Figure 13.3 Buttons incrementing and decrementing the parameter value

As you see, the value of the parameter changes when you click the buttons, but the slider stays at its position, which becomes inconsistent with the parameter value. To fix this, we will have the slider move each time the value of the parameter changes.

Add the On change code of the parameter

7. Navigate to the **Advanced** section of the parameter's properties and type the following code in the **On change** field: **slider.setValue(parameter);**
8. Run the model. Click the buttons and watch the slider.

The parameter value, the slider position, and the enabled/disabled buttons are now all consistent.

Example 13.3: Edit box linked to a parameter of flowchart block

Not only can controls be linked to the parameters and variables of the "current" agent (i.e. the one where they belong to), but also to the parameters of other agents and flowchart blocks. In this example we will link the edit box to the **rate** parameter of the **Source** block in a simple process model.

Follow these steps:

1. Open the **Process Modeling Library** palette and drag the **Source** block onto the graphical editor.

2. Now drag the **Sink** block to the right of the **Source** block. Place it close enough to let the blocks automatically connect.
3. Open the **Controls** palette and drag the **Edit Box**. Place it to the left of the **source** block.
4. Navigate to the properties of the edit box, check the **Link to** checkbox and select **source** from the drop-down list. The **Parameter** drop-down list will appear below with the block's **rate** parameter selected by default.
5. Define the edit box' values range:
   Minimum value: 0
   Maximum value: 100
6. [optional] Using the **Text** element from the **Presentation** palette create an explanatory text "Arrival rate:" to the left of the edit box.
7. Run the model.
8. Click the **source** block to bring up its inspect window.
9. Try to set different values of the arrival rate: 20, 0, -1, "abc", etc.

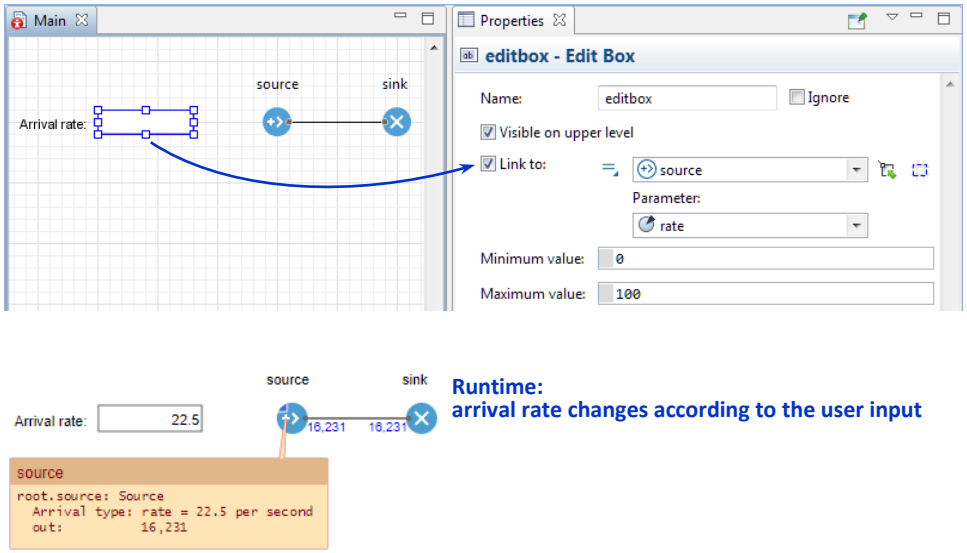


Figure 13.4 Edit box linked to a parameter of Source block

The **rate** parameter of **source** changes with each new valid value entered; the corresponding **set_rate()** function is called each time. In case the edit box is linked to a parameter of numeric type, the invalid inputs, as well as numeric inputs that are out of the given range, are automatically rejected and the value does not change.

Example 13.4: Radio buttons changing the view mode

Let us use radio buttons to change the view mode of a model with the US map as part of the interface. Assume we want to give the user the ability to see three

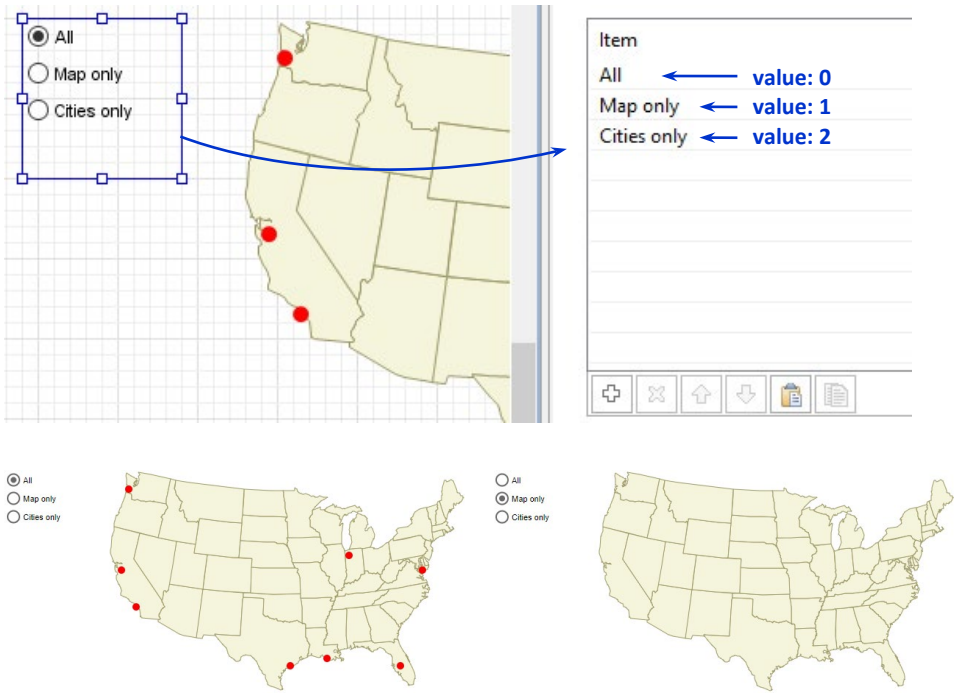
possible configurations: a) the map with main cities, b) the map without cities, or c) the cities without the map.

Follow these steps:

1. Drag the **USA Map** from the **Pictures** palette to the graphical editor.
2. Create a small red circle by dragging the **Oval** element from the **Presentation** palette, adjusting its size to 10 pixels and setting the fill color to red.
3. Place the circle on the Pacific coast in southern California, approximately where Los Angeles is located.
4. Zoom in the editor and Ctrl+drag the circle to create its copies for San Francisco, New York, Orlando, Houston (or any other cities you like).
5. Select all circles (by dragging the selection rectangle), but do not include the map in this selection. If the map gets selected Ctrl+click it to deselect.
6. Right-click one of the circles and choose **Grouping | Create a Group** from the context menu. All circles should now be in one group.
7. Open the **Controls** palette and drag the **Radio Buttons** element to the left of the map.
8. In the properties of the radio buttons specify the following list of choices in the **Item** list: **All**, **Only map**, **Only cities**.
9. Select the map. Click the **Visible** property's icon to switch to the dynamic value editor and type: **radio.getValue() != 2**
10. Select the group of cities by clicking one of the circles. Switch the **Visible** property to dynamic value editor and type: **radio.getValue() != 1**
11. Run the model. Switch radio buttons.

In this example the radio buttons are not linked to any variable, but their function **getValue()** is used to control the shapes visibility via their dynamic properties.

The value of the radio buttons controls is of type **int**, not **String**, and starts at 0.



Runtime: the user is able to switch between different views

Figure 13.5 Radio buttons control the visibility of the map and the cities

Example 13.5: Combo box controlling the simulation speed

We will use combo box to control the speed of simulation. Although there are controls for changing the simulation speed in the model window, the combo box may be useful to provide a limited choice of speeds, which might make sense for a certain model.

Follow these steps:

1. Drag the **Combo Box** element from the **Controls** palette to the graphical editor and extend it as shown in Figure 13.6.
2. In the combo box properties enter the following items in the **Item** list:
   x1
   x10
   Fast
3. Expand the **Action** properties section and type the following code in the field:

```
if( value.equals( "Fast" ) )
  getEngine().setRealTimeMode( false );
else {
```

```
  getEngine().setRealTimeMode( true );
  if( value.equals( "x1" ) )
    getEngine().setRealTimeScale( 1 );
  else if( value.equals( "x10" ) )
    getEngine().setRealTimeScale( 10 );
}
```

4. To view the simulation speed change, create the simplest process model consisting of a **Source** block connected to a **Sink** block (drag both blocks from the **Process Modeling Library** palette and place them close to each other to automatically connect their ports).
5. Run the model. Set different speeds using the combo box. Observe the speed at which the agents are generated.

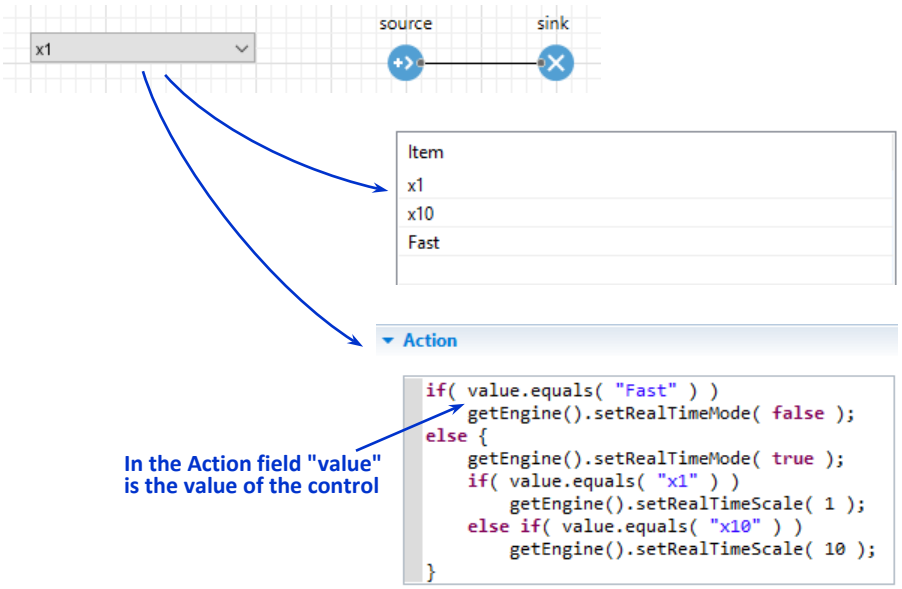


Figure 13.6 Combo box - custom control of the simulation speed

The combo box value is a string, therefore in the **Action** field we need to compare the value to the three different **String** constants.

Example 13.6: File chooser for text files

In this example the **File chooser** will be combined with the **Text file** element so that the user of the model will be able to choose a file (e.g. with the model parameters) before starting the model.

Follow these steps:

1. Drag the **File Chooser** from the **Controls** palette to the graphical editor and extend it a bit as shown in Figure 13.7.

2. Open the **Connectivity** palette and drag the **Text File** element to the right of the file chooser.
3. Open the **Presentation** palette and drag the **Text** element below the file chooser, as shown. Set the font size of the text to 11pt.
4. Navigate to the properties of the file chooser and type **Open a text file** in the **Title** field.
5. In the **File name filters** field type: **.txt**
6. Expand the **Action** section of the file chooser properties and type the following code in the field:

```
file.setFile( value, TextFile.READ );
text.setText( "" );
while( file.canReadMore() )
  text.setText( text.getText() + file.readString() + "\n" );
```

7. Run the model.
8. Click the file chooser button. Select any text file and click **Open**. The file contents will appear on the screen.

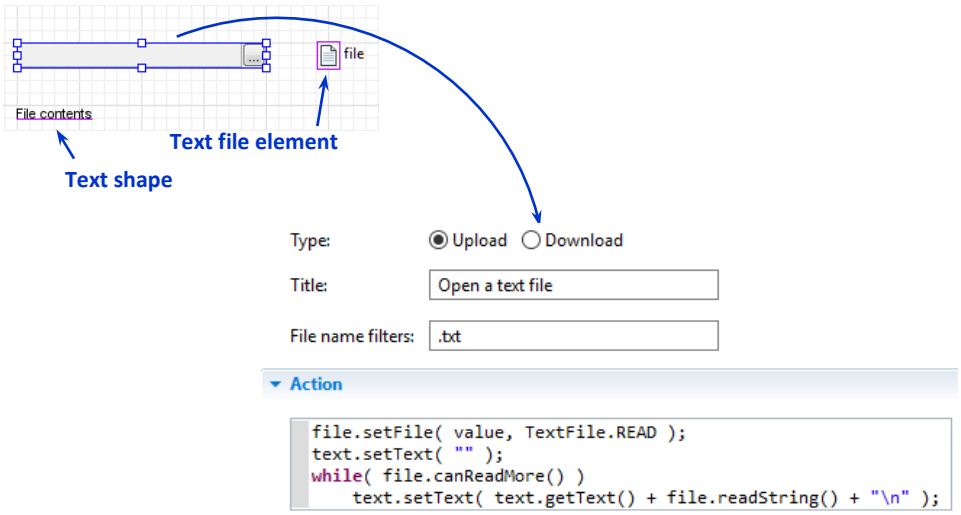


Figure 13.7 File chooser works with the Text file element

The file chooser is set up as follows. The title field becomes the title of the **Open** dialog. The name filter tells the dialog to show only files with the "txt" extension. When the user chooses a file, the file chooser action is executed. It sets up the chosen file to the Text file element **file** – the element that can read and write text files. Then the text of the **text** shape is cleared (we assign an empty string **""**) and the contents of the text file is added to the text shape line by line (**"\n"** symbol denotes the end of line).

Indivisibility of control actions and model events

The actions associated with controls are initiated by the user and may be executed during model runtime. Therefore, you may wonder how they are synchronized with the events in the model.

AnyLogic guarantees indivisibility of all control actions and all model events. It means that the control action code is executed without discontinuities (it is atomic) and is never interrupted by the execution of the model events and vice versa. If the user changes the control state during the continuous phase of the model execution (e.g. when the dynamic equations are being solved numerically), the control action is treated as yet another discrete event, so the solver correctly stops before the control action and resumes after.

13.1. Dynamic properties of controls

Just as is done for shapes, the dynamic properties of controls are constantly evaluated at runtime and may be used to dynamically change control availability, visibility, size, position, etc.

Almost all controls have the **Enabled** property. If you enter a Boolean expression there, the control will be enabled if it evaluates to **true**, and disabled otherwise (see Example 1.2: "Buttons changing the parameter value"). The **Visible** property located in the **Advanced** section is used to temporarily hide controls. The size properties are used rarely as typically you do not want to resize the controls at runtime.

Example 13.7: Radio buttons enabling/disabling other controls

You can create sophisticated “dialog-like” behaviors by binding the **Visible** and/or **Enabled** properties of some controls to the states of other controls. Suppose you want to suggest two modes to the user: the use of default parameters or a custom setup. You can use radio buttons to enable/disable controls linked to the model parameters.

▶ **Follow these steps:**

1. Create a group of two radio buttons by dragging the **Radio buttons** element from the **Controls** palette.
2. In the **Item** table in the radio buttons properties set the labels of the two buttons to “Use default settings” and “Use custom settings” respectively.
3. Create a slider below the radio buttons as shown in Figure 13.8. You may need to adjust the radio buttons size to avoid overlapping with the slider.
4. Create a parameter to the right of the slider. Set the default value of the parameter to **50**.

5. In the slider properties, select the **Link to** checkbox and select **parameter** from the drop-down list on the right.
6. Switch the **Enabled** property to the dynamic value editor and type: `radio.getValue() == 1`
7. Expand the **Action** section of the radio buttons properties and type the following code in the field:

```
if( value == 0 )
  set_parameter( 50 );
else
  set_parameter( slider.getValue() );
```

8. Run the model. Play with radio buttons and the slider. Note how the parameter value changes.

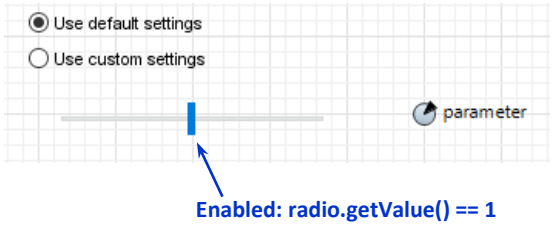


Figure 13.8 Radio buttons enable and disable the slider

Note that the slider remembers its position while the radio buttons are switched to the first option. Therefore, when the user switches to the second option, we force the parameter value to the slider value.

Example 13.8: Replicated button

Sometimes you need to create an array of similar controls e.g. to be able to change an array of similar elements at runtime. AnyLogic *replicated controls* may save you some drawing time and it may also make your model scalable. In this example we will create a replicated button and use it to change the fill color of a replicated shape. Moreover, the number of copies of the button and the shape will be dynamically controlled by a slider.

Follow these steps

1. Create the following four elements as shown in Figure 13.9: a slider, a variable, a button and a rounded rectangle.
2. Set the name of the variable to **N**, its type to **int**, and initial value to **3**.
3. Link the slider to the variable **N** and set its minimum and maximum values to **1** and **9** respectively.
4. In the **Advanced** section of the rounded rectangle properties, type **N** in the **Replication** field.

5. In the **Position and size** section, switch **Y** to the dynamic value editor and type: **100 + 50 * index**
6. Navigate to the properties of the button and type **N** in the **Replication** field of the **Advanced** section.
7. Switch the **Label** property to the dynamic value editor and type: **"Paint shape " + index**
8. In the **Position and size** section of the properties, switch **Y** to the dynamic value editor and type: **100 + 50 * index**
9. Expand the **Action** section of the button properties and type the following code in the field: **roundRectangle.get(index).setFillColor(blueViolet);**
10. Run the model. Move the slider and click the buttons.

As you can see, the number of buttons changes as the slider changes the variable **N**. By using the index of the button copy in the button's **Action** field you can do different things with different buttons, in this case - change the color of the shape whose number equals the number of the button. The label of the button also depends on the index.

If you paint the shape with a number, let's say 8, then set the number of shapes to 5 and back to 9, the new 8th shape will be not painted because it is a brand new object. The old one has been completely deleted.

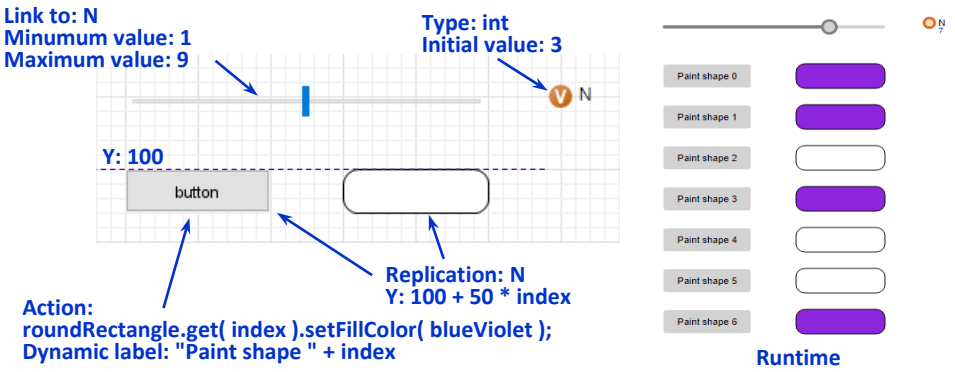


Figure 13.9 Replicated button controlling a replicated shape

13.2. Controls' API

Just like anything in AnyLogic, the controls are mapped to Java objects and expose their API (application program interface) to the modeler. In many cases the same effect can be achieved by using either the dynamic properties of a control or calling its functions.

The most frequently used functions of controls are:

- **setVisible(boolean)** – hides or shows the control
- **setEnabled(boolean)** – enables or disables the control
- **action()** – executes the action of the control
- **getValue()** – returns the current value of the control
- **setValueToDefault()** – sets the value or the control to the default one
- **setValue(..., boolean)** – sets value to a given one and optionally calls the action

The full list of functions is available in *AnyLogic Help. Advanced Modeling with Java: API reference* (The AnyLogic Company, 2019). In Figure 13.10 below we give the Java class hierarchy for controls. Note that the base class for all controls is **ShapeControl**, which is a subclass of **Shape**, therefore controls implement the functions of **Shape**. The class **ShapeTextField** corresponds to the edit box control.

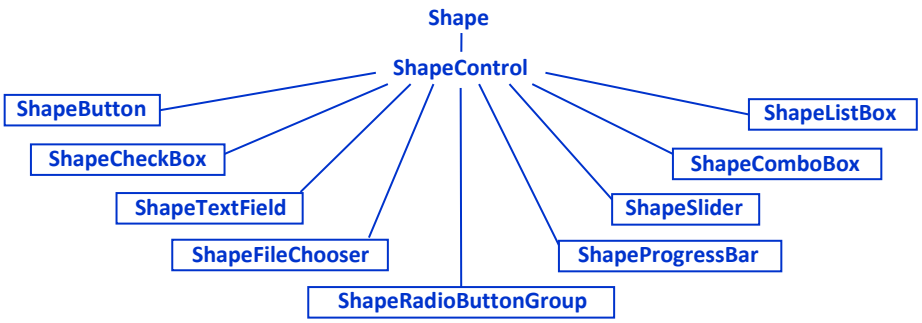


Figure 13.10 AnyLogic Java classes for controls

13.3. Handling mouse clicks

Handling mouse clicks is yet another way to add interactivity to your model. You can use click handling to display additional information on the model elements, to create hyperlinks, to define locations on the map, to control specific agents, and so on.

Mouse clicks in the model window are processed as follows. AnyLogic iterates through all shapes in their Z-order, starting from top. If a shape area contains the coordinates of the click and the shape's **On click** action is defined, the action is executed (and, by default, returns **false**). If the action returns **true** (which you need to do explicitly), the click processing will stop. Otherwise the iteration continues down to the last shape at the bottom, see Figure 13.11.

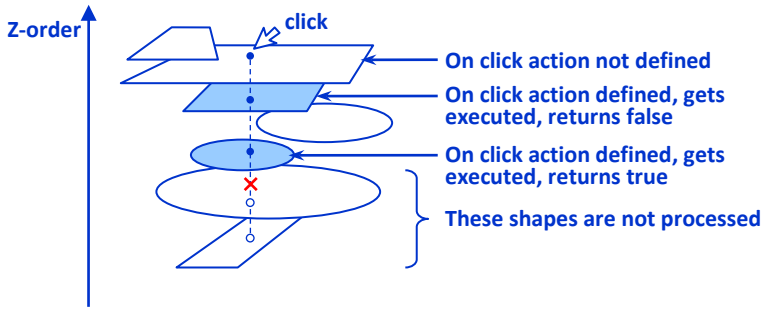


Figure 13.11 AnyLogic processing of mouse clicks

In the **On click** code field you can also access the exact coordinates of the click relative to the shape coordinates. They are available as **clickx** and **clicky**.

Example 13.9: Hyperlink menu to navigate between view areas

Click handling is often used to provide custom navigation in the model. Modelers create text or graphics and make them hyperlinks to various important locations. In this example we will show how to create a simple hyperlink menu to switch between two locations marked with view areas. An area may contain the model animation while another might contain the model output.

Follow these steps:

1. Create a view area at the coordinate origin of the **Main** agent type. The **View Area** element is located in the **Presentation** palette.
2. In the properties of the view area change:
   Name: **viewAnimation**
   Title: **Animation**
3. Create another view area at (0,600).
4. In the properties of the second view area set:
   Name: **viewOutput**
   Title: **Output**
5. Draw a circle or any other graphics in the center of the **Animation** view, i.e. at (500, 300) just to identify the view at runtime.
6. Drag a time plot or any other chart from the **Analysis** palette in the center of the **Output** view, i.e. at (500, 900) for the same purpose.
7. Create two texts (**Text** presentation shapes) “Animation” at (50,20) and “Output” at (150,20). Set their font size to **16 pt**, **Bold**.
8. Set the color of the text “Output” to **blue**.
9. In the **Advanced** properties section of the text “Output” write the following code in the **On click** field: **viewOutput.navigateTo();**

10. Draw a blue line under the text "Output" to make it look like a hyperlink, see Figure 13.12.
11. Select both texts "Animation" and "Output", and the blue line. Ctrl+drag the selection to create a copy.
12. Drag the copy downwards to the second view area until the text "Animation" is at (50, 620).
13. Set the color of the second "Animation" text to blue, and "Output" – to black.
14. Move the blue line from "Output" to "Animation" and extend it to match the text.
15. Cut the code from the **On click** field of the **Advanced** properties section of the second "Output" text to the same field of the "Animation" text and change it to: **viewAnimation.navigateTo();**
16. Run the model. Click the hyperlinks you created.

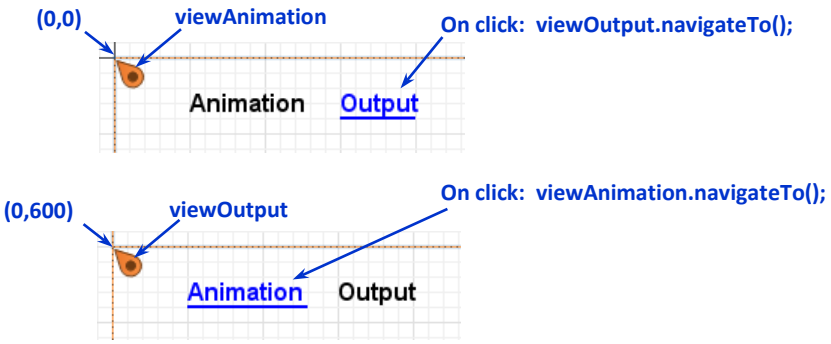


Figure 13.12 Hyperlink navigation between view areas

The blue texts in this example are click-sensitive. Their **On click** actions call the function **navigateTo()** of the two view areas, which displays the corresponding part of the presentation.

If you do not care how the click is further processed (as in the case where there are no shapes below the click-sensitive ones, or those shapes are click-insensitive), you can omit the **return** statement at the end of the **On click** code – just as we did it in this example.

Example 13.10: Creating dots at the click coordinates

You can not only find out that a certain shape was clicked, but also the exact coordinates of the click (relative to the shape). In this example we will use that to create small dots at the locations of the clicks.

Follow these steps:

1. Create a rectangle with the upper left corner at approximately (50,50) and size 500 x 500.
2. Name it **clickArea**.
3. In the **Advanced** section of the rectangle properties type the following code in the **On click** field:

```
ShapeOval dot = new ShapeOval();
dot.setRadius( 2 );
dot.setFillColor( blue );
dot.setLineColor( null );
dot.setPos( self.getX() + clickx, self.getY() + clicky );
presentation.add( dot );
```

4. Run the model. Click within the rectangle bounds.

The **clickx** and **clicky** variables available in the **On click** field are the coordinates of the click *relative to the shape that catches the click*. To transform them to the absolute coordinates we need to add the coordinates of the shape itself.

Certain elements provide easy access to themselves via the variable **self** in the action fields (the **On click** field of all presentation shapes, **Action** of all controls except progress bar, etc.). Using **self** instead of the element name may help you make the code element-independent and reusable.

Example 13.11: Catching mouse clicks anywhere in the model window

You already know that mouse clicks are handled by shapes. What if you need to catch mouse clicks anywhere in the model window? One of the ways to do it is to create a very large invisible shape that handles the clicks.

Follow these steps:

1. Using the **Oval** element from the **Presentation** palette, create a small circle anywhere on the graphical diagram.
2. Leave the default name **oval** and set the radius to 5 pixels. We will use that circle to show the click location.
3. Add a rectangle. Enlarge it to cover all the area you expect to use for the model.
4. In the **Appearance** section of the rectangle properties, choose **No color** in both the **Fill color** and the **Line color** controls.
5. In the **Advanced** properties section, type the following code in the **On click** field:

```
clickx += self.getX();
clicky += self.getY();
```

```
oval.setPos( clickx, clicky );
```

6. Run the model. Click in different places.

Our rectangle covers all the meaningful area of the model presentation. To make the shape click-sensitive we define its **On click** action – in this example it places the circle at the click location.

Note that you will not be able to select the invisible shape in the graphical editor. In case you need to edit it, select the shape in the **Projects** tree (you will find the rectangle in the **Main** agent's **presentation** branch).