Appendix A
Reference: Built-in Methods

The Methods panel has three tabs so as to distinguish between *procedural* methods, *functional* methods, and methods related to that object's specific *properties*. Figure A.1 illustrates the three tabs in a side-by-side listing, using *penny* (a **Penguin** object) as an example.

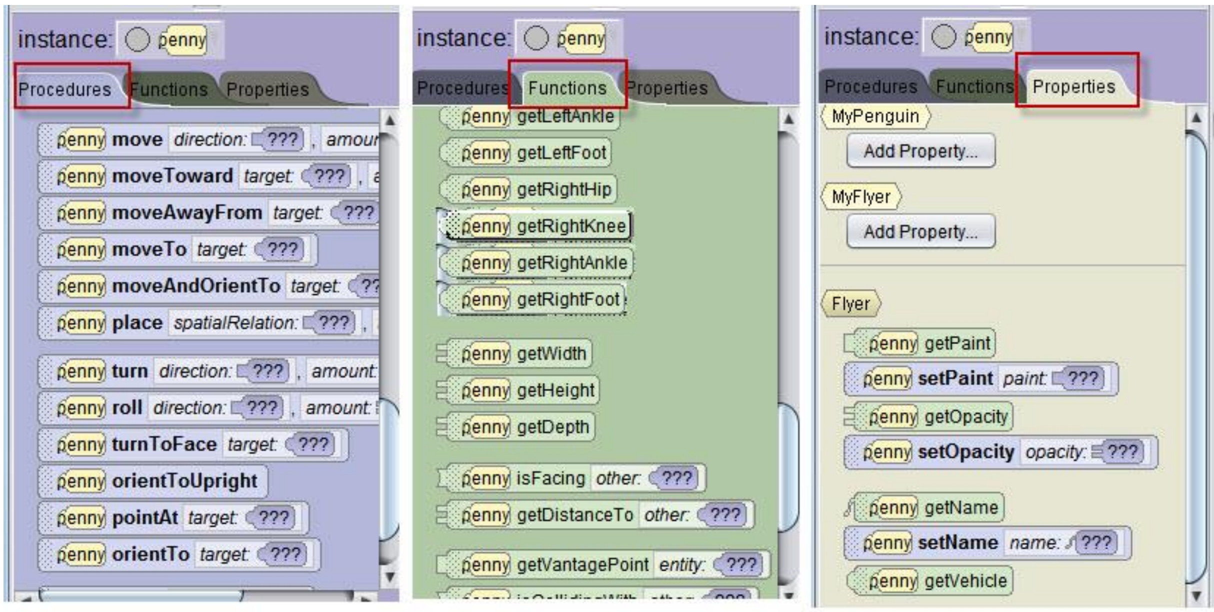


Figure A.1 Side-by-side listing of built-in procedural, functional, and property methods

Important concepts:

Procedural methods describe actions that may be performed by an object, such as move, turn, or roll. These actions often change the location and/or orientation of an object. The important thing to know about procedural methods is that they each perform an action but do not compute and return an answer to a question.

Functional methods are expressions that compute and answer a question about an object such as what is its width or height, or what is its distance from another object.

Properties methods are methods for retrieving (get) and changing (set) specific properties of an object of this class. These specific properties, such as paint, opacity, name, and vehicle, are used in animation rendering.

As a convenient reference, the remainder of this Chapter describes the method tiles commonly found in the Procedures, Functions, and Properties tabs for an object in a scene. The Figures and examples use the alien object, as seen in the screenshot in Figure A.2.

Figure A.2

PROCEDURAL METHODS

Change the size of an object

Every object in Alice has three dimensions, all having a height, width, and depth (even if the value of that dimension is 0.0; e.g., a disc may have a height of 0.0). These procedures change the size of an Alice object, by changing all the dimensions at the same time, proportionately. Procedures that change the value of height, width, or depth are shown in Figure A.3 and summarized in Table A.1.

Figure A.3 Procedures that change the size of an object

The *set* procedures change that dimension to the absolute size provided in the statement. For example if the alien has a height of 1.5 meters, the statement

alien.setHeight *height: 2.0*

will animate the alien growing to a height of 2.0 meters. The value 2.0 is an **argument** to the method, to be used as the targeted height.

The *resize* procedures change a dimension by the factor of the argument value provided in the statement. For example if the same alien has a height of 1.5 meters, the statement

alien.resizeHeight ***factor: 2.0***

will animate the alien growing to the height of 3.0 meters, as the height of the alien is increased by a factor of 2.

Table A.1 Procedures that change the size of an object

Procedure	Argument(s)	Description
setWidth	*DecimalNumber*	Changes the value of the object's width to the value of the argument *width*, with width and depth changed proportionately.
setHeight	*DecimalNumber*	Changes the value of the object's height to the value of the argument *height*, with height and depth changed proportionately.
setDepth	*DecimalNumber*	Changes the value of the object's depth to the value of the argument *depth*, with height and width changed proportionately.
resize	*DecimalNumber*	Changes all the dimensions of the object by the value of the argument *factor*, proportionately
resizeWidth	*DecimalNumber*	Changes the width dimension of the object by the value of the argument *factor*, with height and depth changed proportionately.
resizeHeight	*DecimalNumber*	Changes the height dimension of the object by the value of the argument *factor*, with width and depth changed proportionately.
resizeDepth	*DecimalNumber*	Changes the depth dimension of the object by the value of the argument *factor*, with height and width changed proportionately.

Change the position of an object in the scene

Every object in Alice has a specific position and orientation in the scene. Each object can move to its *left* or *right*, *forward* or *backward*, *up* or *down*. Procedures that change an object's position are shown in Figure A.4 and summarized in Table A.2.

Figure A.4 Procedures that change the position of an object in the scene

Table A.2 Procedures that move an object to a different position in the scene

Procedure	Argument(s)	Description
move	*Direction, DecimalNumber*	Animates movement of the object in the specified *direction* according to its own orientation, by the specified *amount*
moveToward	*Model, DecimalNumber*	Animates movement of the object, by the specified *amount*, in the direction of the *target* object (a 3D Model)
moveAwayFrom	*Model, DecimalNumber*	Animates movement of the object, by the specified *amount*, directly away from the position of the *target* object (a 3D Model)
moveTo	*Model*	Animates movement of the object, in the direction of the *target* object (a 3D Model) until the pivot point of the object and the pivot point of the target are exactly the same; the original orientation of the object is unchanged.

moveAndOrientTo	*Model*	Animates movement in the direction of the *target* object (a 3D Model) until the pivot point of the object and the pivot point of the target are in exactly the same position and the orientation of the object is the same as the orientation of the target object.
place	*spatialRelation:* ABOVE, BELOW, RIGHT_OF, LEFT_OF, IN_FRONT_OF, BEHIND; *Model*	Animates movement of the object, so that it ends up 1 meter from the *target* object (a 3D Model) along the specified *spatialRelation*

Change the orientation of an object in the scene

Every object in Alice has a specific orientation in the scene, with its own sense of *forward* and *backward*, *left* and *right*, *up* and *down*. Importantly, each object has a *pivot* or *center* point, around which these rotations occur. Procedures that change an object's position are shown in Figure A.5 and summarized in Table A.3.

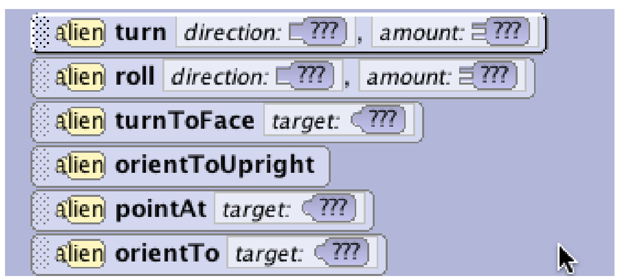


Figure A.5 Procedures that rotate an object

Turn rotations can be LEFT, RIGHT, FORWARD, or BACKWARD. Roll rotations can only be LEFT or RIGHT. The rotations occur in **the direction of an object's own orientation, not the camera's point of view and not as seen by the viewer of the animation.** For example, if an object is given an instruction to turn LEFT, the object will turn to its own left (which may or may not be the same as left for the person viewing the animation).

The amount of a rotation is always described as a fractional part of a full rotation, expressed as a decimal value. For example, the statement

alien.turn *direction: RIGHT, amount: 0.25*

will animate the alien turning to its right ¼ of a full rotation, expressed as 0.25. Although a full rotation is 360 degrees and ¼ rotation is 90 degrees, Alice does not use degrees to specify the rotation amount. So, always convert any amount in degrees to a fractional part of a rotation, expressed as a decimal value.

Generally a *turn* will result in an object's sense of *forward* changing as the animation occurs, although it may come back to its original orientation if it turns all the way around. A *roll* will result in an object's sense of *up* changing as the animation occurs, although it may come back to its original orientation if it rolls all the way around. It may be helpful to note that an object's sense of forward stays the same during a roll.

Table A.3 Procedural methods that rotate an object

Procedure	**Argument(s)**	**Description**
turn	*Direction, DecimalNumber*	Animates a turn of an object around its pivot point, in the specified *direction* according to its own orientation, by the specified *amount*, given in fractional parts of a rotation. The object's sense of **forward** will be changing during the animation
roll	*Direction, DecimalNumber*	Animates a roll of the object around its pivot point, in the specified *direction* according to its own orientation, by the specified *amount*, given in fractional parts of a rotation. The object's sense of **forward** will remain unchanged during the animation
turnToFace	*Model*	Animates a turn of the object around its pivot point, so that its sense of forward will be in the direction of the *target* (a 3D Model object)
orientToUpright		Animates a rotation of the object around its pivot point, so that its sense of up will be perpendicular to the *ground*
pointAt	*Model*	Animates a rotation of the object around its pivot point, so that its sense of forward will be in the direction of the *target's* (a 3D Model object) pivot point
orientTo	*Model*	Animates a rotation of the object around its pivot point, so that its orientation will be exactly the same as the orientation of the target (a 3D Model object). The object's position will be unchanged.

Other procedures

Some procedures do not neatly fit into the descriptive categories of the preceding paragraphs. We have collected these procedures into a category called "Other." These procedures provide program output (*say*, *think*, *playAudio*), manage timing in an animation (*delay*), simplify returning an object to its original position after an animation (*straightenOutJoints*), and allow one object to be the vehicle for another object as it moves around the scene (*setVehicle*). The Other procedures are shown in Figure A.6 and summarized in Table A.4.

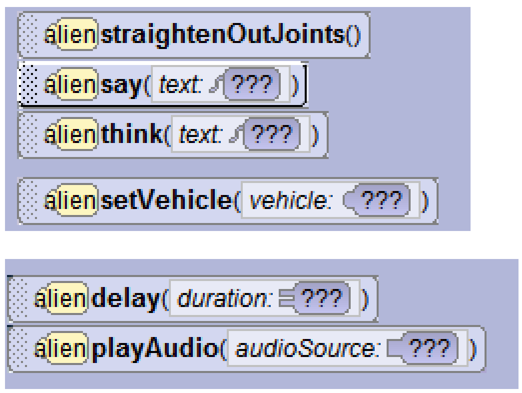


Figure A.6 Other procedures

Table A.4 Other procedures

Procedure	Argument(s)	Description
straightenOutJoints		Restores all the joints of *this* object to their original position, when *this* object was first constructed in the scene editor
say	*textString*	A speech bubble appears in the scene, containing the value of the *text* argument, representing something said by *this* object
think	*textString*	A thought bubble appears in the scene, containing the value of the *text* argument, representing something thought by *this* object
setVehicle	*Model*	Any movement or rotation of the *target* (a 3D Model object) will produce a corresponding movement by *this* object. *This* object cannot be a vehicle for itself, and two objects may not have a reciprocal vehicle relationship (in other words, *this* object cannot be the vehicle of the *target* object, if the *target* object is

		already the vehicle for *this* object)
delay	*DecimalNumber*	The animation pauses for the length of the *duration* in seconds
playAudio	*??? (sound file)*	The entire imported sound file (either .mp3 or .wav format) will be played in the animation. The length of sound clip that is actually played can be modified in **AudioSource** drop-down menu and selecting ***Custom Audio Source…*** See **Chapter 5: How to…**

Important concepts:

Do in order

When a *delay* action is performed within a *Do in order,* Alice waits the specified number of seconds before proceeding to the next statement. Calling a *delay* on the scene will suspend the animation until the *delay* is complete.

When a *playAudio* action is performed within a *Do in order,* Alice plays the sound for the specified amount of time before proceeding to the next statement.

Do together

When a *delay* action is performed within a *Do together*, other statements within the *Do together* are not affected. However, the *delay* does set a minimum duration for execution of the code block within the *Do together*. For example, in the code block shown below, the alien will move and turn at the same time (duration of 1 second), but Alice will not proceed to the statement following the *Do together* until the *delay* is completed (2 seconds).

```
Do together
    alien.turn direction: RIGHT, amount: 0.25
    alien.delay duration: 2.0
    alien.move direction: FORWARD, amount: 1.00

bunny.turn direction: LEFT, amount: 1.0
```

When a *playAudio* action is performed within a *Do together,* Alice starts plays the sound at the same time as other statements within the *Do together* are executing (for example, as background music).

Most procedures in Alice have a set of parameters with default argument values. These are known as *detail* parameters. The detail parameters enhance or fine tune the animation action performed when a statement is executed.

Figure A.7

The three most common detail parameters are *asSeenBy*, *duration*, and *animationStyle*. There are a few procedures that may not use all of these details, or they may have a different set of details, appropriate for that particular animation. Table A.5 summarizes the detail parameter options.

Table A.5 Details

Detail	Values	Description
asSeenBy	*Model*	The movement or rotational animation of this object will be as if *this* object had the pivot point position and orientation of the *target* object
duration	*DecimalNumber*	By default, Alice animation methods execute in 1 second. This modifier changes the duration value to a specified length of time.
animationStyle	*BEGIN_AND_END_ABRUPTLY* *BEGIN_GENTLY_AND_END_ABRUPTLY* *BEGIN_ABRUPTLY_AND_END_GENTLY* *BEGIN_AND_END_GENTLY*	The default animation style is *BEGIN_AND_END_GENTLY*, which begins with a reasonable period of acceleration, then constant movement at some top speed, followed by a reasonable period of deceleration. Other animation styles: *BEGIN_GENTLY_AND_END_ABRUPTLY* begins with a *gradual* acceleration to top speed and ends with a sudden

		stop. *BEGIN_ABRUPTLY_AND_END_GENTLY* starts at top speed and ends with gradual deceleration. *BEGIN_AND_END_ABRUPTLY* starts at top speed and ends with a sudden stop.

FUNCTIONAL METHODS

Functions that provide access (a link) to an internal joint of an object

The internal joints of an object are part of a skeletal system. For this reason, a function is called to access an individual joint within the skeletal system. These functions return a *link* to the joint (similar to a link that holds the address of a web page on the web).

As an example, some of the functions to access the individual joints of an alien object are illustrated in Figure A.8 accompanied by an X-ray view of the alien's internal joints. (NOTE: Due to page space limitations, not all the alien's joint access functions are listed here.)

Figure A.8 Functions that link to an internal joint of an object

The link returned by calling one of these functions provides access to the specified joint of the object, for example, if in an animation we wanted a ball to move to the alien's right hand in a game of catch with another alien, we could write the instruction statement:

ball.moveTo *target:* **alien.getRightHand**

It should be noted that these functions are dependent upon the design in the 3D Model for which the object is constructed. For example, all Bipeds have the same basic set of joints, as shown in the X-ray view of the alien and hare in Figure A.9.

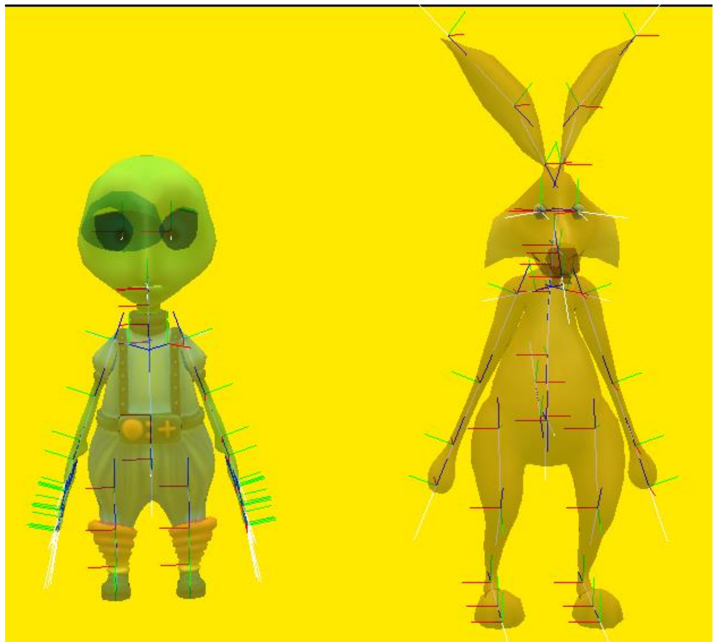


Figure A.9 X-ray view of alien and hare internal joints

Although the alien and the hare have the same basic set of **Biped** joints, the alien also has a set of finger joints that are particular to the **Alien** class and the hare has a set of joints in its ears that are specific to the **Hare** class. These commonalities and differences are reflected in the functional methods that get access to an internal joint, as shown in Figure A.10.

Figure A.10 Common and specific functional methods for joint access

Getters: Functions that return the dimension values of an object

The term "getter" is used to describe a function that returns the current value of a property. In Alice the three dimension (width, height, and depth) properties are of special importance and have their own getter functions. These getter functions for the alien are shown in Figure A.11. Table A.6 summarizes these functions.

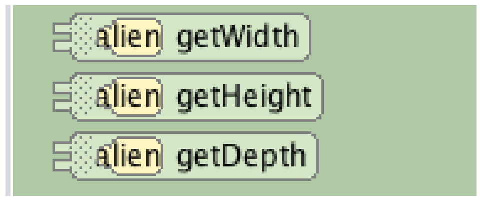


Figure A.11 Functions that return dimension property values

Table A.6 Functions that return dimension values

Function	**Return type**	**Description**
getWidth	*DecimalNumber*	Returns the width (left to right dimension) of *this* object
getHeight	*DecimalNumber*	Returns the height (bottom to top) dimension of *this* object
getDepth	*DecimalNumber*	Returns the depth (front to back) dimension of *this* object

Other functions

Some functions do not neatly fit into the descriptive categories of the preceding paragraphs. We have collected these functions into a category called "Other." Other functions are shown in Figure A.12 and summarized in Table A.7.

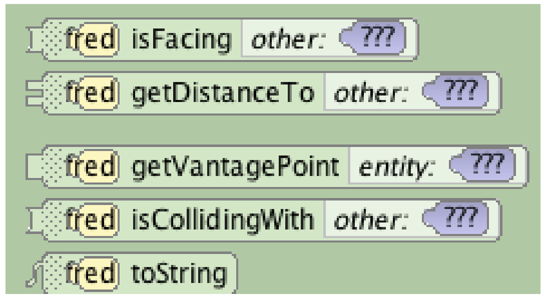


Figure A.12 Other functional methods

Table A.7 Other functional methods

Function	Return type	Arguments	Description
isFacing	*Boolean*	*Model*	Returns **true** if *this* object is facing the *other* (a 3D Model object) or else returns **false**
getDistanceTo	*DecimalNumber*	*Model*	Returns the distance from the center point of *this* object to the center point of the *other* (a 3D Model object)
getVantagePoint	*???*	*entity*	TO BE IMPLEMENTED. Returns the point of view of *this* object
isCollidingWith	*Boolean*	*Model*	returns true if the bounding box of *this* object intersects in any with the bounding box of the other (3D Model object), false otherwise
toString	*TextString*		NOTE: THIS DOES NOT RETURN THE IDENTIFIER NAME OF THIS OBJECT IN PROGRAM CODE, but the internal identifier used by Alice in the virtual machine

Functions for User Input

Functions that ask the user to use the keyboard or mouse to enter a value (of a specific type) are provided for all objects in an Alice scene. The value entered by the user is returned by the function, to be stored in a variable or used as an argument in a call to another procedure or function. Functions for User Input are shown in Figure A.13.

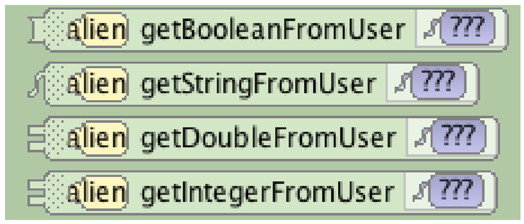


Figure A.13 Functions for User Input

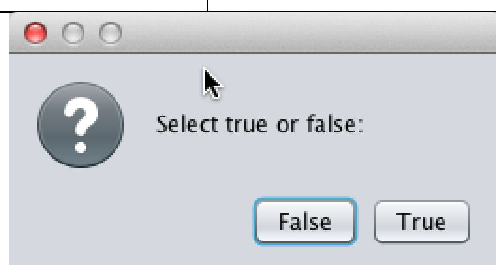
When a user input function is called, at runtime, a dialog box is displayed containing a prompt to ask the user to enter a value of a specific type. Of course, Alice does not automatically know what prompt to use in the dialog box. The programmer supplies a prompt that will be displayed. The prompt is the argument to the function within an instruction statement.

An important part of calling a function to get user input is that the value the user enters is expected to be of a specific type. For example, the value the user enters when the *getIntegerFromUser* function is called must be a whole number, not a number containing a decimal or a fraction. Likewise, a variable or a parameter that receives the returned value must be a compatible type with the type of value being returned by the function. For example, if the user enters a String of alphabetic characters, the String cannot be stored in an Integer variable. For this reason, Alice will continue to display the user input dialog box until the user enters a value of the right type.

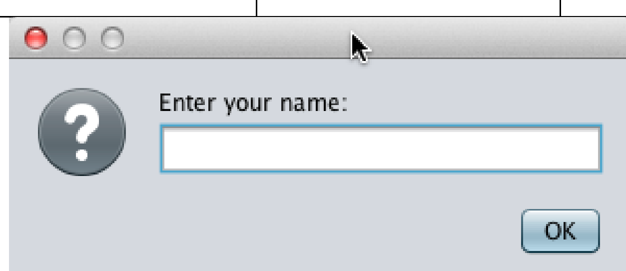
To each row of Table A.8, we have attached an image depicting a sample dialog box containing a prompt appropriate as an argument for calling that function.

Table A.8 User input Table

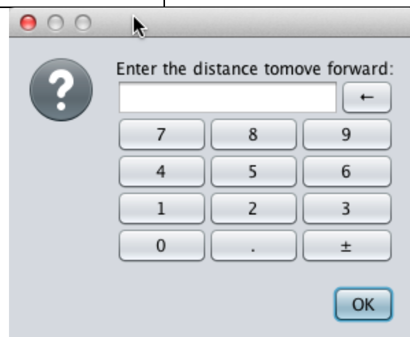
Function	Return type	Argument	Description
getBooleanFromUser	*Boolean*	*TextString*	Displays the dialog box with the TextString argument displayed as the prompt and True and False buttons for user input.



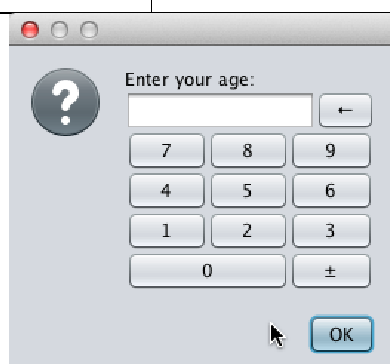
getStringFromUser	*TextString*	*TextString*	Displays the dialog box with the TextString argument displayed as the prompt and a textbox for user input.



getDoubleFromUser	*DecimalNumber*	*TextString*	Displays the dialog box with the TextString argument displayed as the prompt and a keypad (with a decimal point) for user input.



getIntegerFromUser	*WholeNumber*	*TextString*	Displays the dialog box with the TextString argument displayed as the prompt and a keypad for user input.



PROPERTY METHODS

Setter is a specialized term used to describe a procedure that changes the value of an object's property. *Getter* is a specialized term used to describe a function that returns the current value of an object's property. Currently in Alice 3, most setters and getters can be found in the Procedures and Functions tabs of the Methods Panel. (For example, *setVehicle* is in the Procedures tab, and *getWidth* is in the Functions tab.)

Some properties, however, are general purpose in that they are defined for the purpose of rendering an object in the scene. Getters and setters for these properties are conveniently listed in the Properties tab of the Methods panel. For example, the alien's setters and getters are shown in Figure A.14 and summarized in Table A.9.

Figure A.14 Getters and setters for specialized properties

Table A.9 Setters and Getters for specialized properties

Procedure	Argument(s)	Description
setPaint	*paint*	Sets the paint value of *this* object to the *paint* argument
setOpacity	*opacity*	Used to set the transparency of *this* object by setting the opacity value of *this* object using a range of values from **0.0** (invisible) to **1.0** (fully opaque).
setName	*name*	NOTE: THIS DOES NOT CHANGE THE IDENTIFIER NAME OF THIS OBJECT IN PROGRAM CODE, but does change the internal identifier used by Alice for debugging purposes.

Function	Return type	Description
getPaint	*paint*	Returns the paint value of *this* object
getOpacity	*DecimalNumber*	Returns the opacity value in the range of **0.0** (invisible) to **1.0** (fully opaque).of *this* object
getName	*TextString*	NOTE: THIS DOES NOT RETURN THE IDENTIFIER NAME OF THIS OBJECT IN

		PROGRAM CODE, but the internal identifier used by Alice in the virtual machine.
getVehicle	*Model*	Returns a link to another object in the scene that is serving as the vehicle for *this* object

Methods that can be called on an object's internal joints

Procedures

As described previously in Chapter 3, almost all 3D model classes in the Gallery have a system of internal joints. The joints can be thought of as the pivot points of *sub-parts* of the object and can be used in the Scene editor to position sub-parts during scene setup. An object's joints are also objects, and program statements can be written to animate an object's sub-parts by rotating and orienting an object's internal joints. Procedures that can be used to animate joints are shown in Figure A.15.

Figure A.15 Procedural methods for an object's internal joints

These procedures perform the same actions that were described for the entire object, but the pivot point is at the joint. For example, a statement can be created to tell the alien to turn its *right shoulder joint backward*, as shown in Figure A.16. As the right shoulder joint turns, the right upper arm, lower arm, and hand also turn. That is, the arm parts are attached to the body through the shoulder joint. For this reason, the arms parts turn when the joint turns.

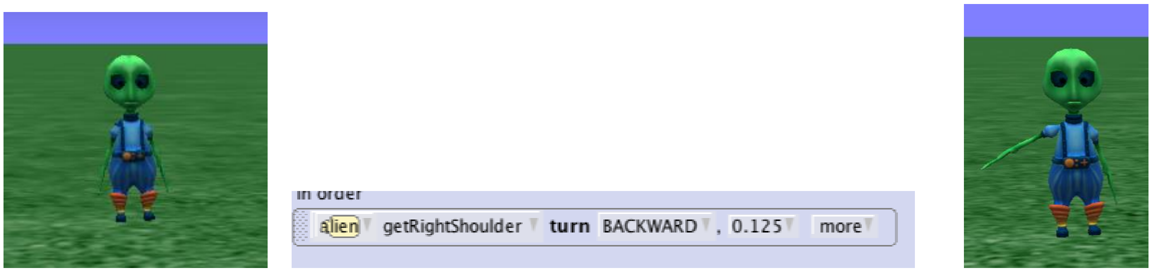


Figure A.16 A statement to turn the alien's right shoulder joint

Notice that the procedures in Figure A.15 do not include methods that *move* the joint. In Alice 3, a joint cannot be moved out of its normal position within the skeletal structure of the object's body. In other words, a joint and its attached sub-part(s) cannot be separated from the body.

The only unique procedure for joints is *setPivotVisible*, as described in Table A.10.

Table A.10 Procedure specific to internal joints

Procedure	Argument(s)	Description
setPivotVisible	**true** or **false**	Displays the pivot position and orientation of *this* joint in the animation if the argument is true, hides the pivot position and orientation of *this* joint in the animation if the argument is false

Functions

Almost all functional methods for an entire object are functions that access (return a link to) one of the joints belonging to that object. However, there are only a few functions that can be called on an individual joint, as shown in Figure A.17.

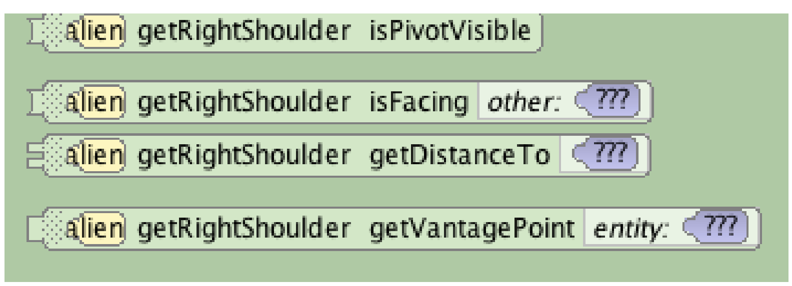


Figure A.17 Functional methods for a joint

The available functional methods have the same name and perform the same actions as the functions of the same name for the entire object. Refer back to Table A.7 for the descriptions of these methods. The only function that is unique to joints is the *isPivotVisible* function, as summarized in Table A.11.

Table A.11 A unique function for internal joints

Function	Return Type	Description
isPivotVisible	*Boolean*	Returns **true** if the pivot position and orientation of *this* joint in the animation is being displayed, or else returns **false** if the pivot position and orientation of *this* joint in the animation is not being displayed

Properties

All of the available getters and setters on the Properties tab/Methods panel of an object's internal joints are the same as the getters and setters of the same name for the entire object, as shown in Figure A.18. Refer to Table A.9 for descriptions of these specialized methods.

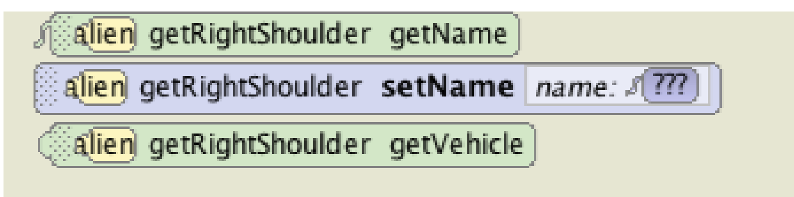


Figure A.18 Properties methods for internal joints

METHODS FOR STANDARD OBJECTS

Every Alice project has a scene (*this*) that is an instance of the **Scene** class and contains two other standard objects: the ground or water surface (an instance of the **Ground** class), and the *camera* (an instance of the **Camera** class), as shown in Figure A.19. Each of these objects has their own procedures, functions, and properties, as defined in their respective classes.

Figure A.19 The standard components of every Alice project

The **Scene** class has a few procedures, functions and property methods that are exactly the same as in other classes, as shown in Figures A.20 A.21, and A.22. See previous descriptions of these procedures (Table A-4), functions (Table A-7), and properties (Table A-9) earlier in this Appendix.

Figure A.20 Procedural methods in common with other classes

Figure A.21 Functional methods in common with other classes

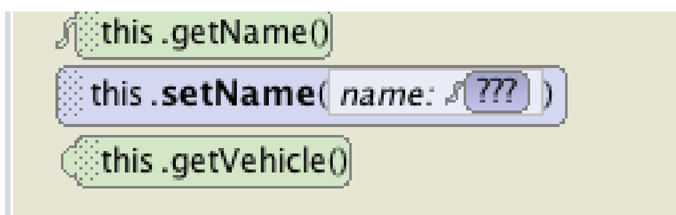


Figure A.22 Properties methods in common with other classes

The scene is truly the "universe" of an Alice 3 project because it provides the stage, the actors, and the scenery for animation. For this reason, a scene object has need of many special methods that perform unique operations for creating the scene and animating the characters in the story or game. Unique procedures that are used for setting up a scene and managing the animation are shown in Figure A.23.

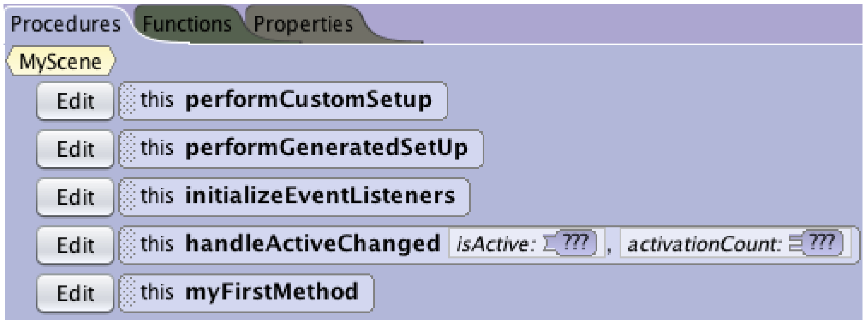


Figure A.23 Unique procedural methods defined in Scene

The Alice environment automatically calls the *performGeneratedSetUp*, *performCustomSetup*, and *initializeEventListeners* procedures (in order) when the user clicks on the ***Run*** button. The *performGeneratedSetUp* procedure contains instructions that were automatically "recorded" as objects were created and arranged in the Scene editor. When *performGeneratedSetUp* is executed, these instructions are used by the Alice system to re-create the scene in the runtime window. The *performCustomSetup* procedure contains instructions that may have been written to adjust the scene in a way not available in the Scene editor. The *initializeEventListeners* procedure contains instructions to start listeners for events such as key presses and mouse clicks while the animation is running. (Specific events and listeners are described below in the Scene Listeners section.)

After these three procedures are executed, the scene's *myFirstMethod* is called and the animation code in the project is executed. Table A.12 provides further information regarding these unique procedural methods.

Table A.12 Procedures for *this* scene

Procedure	**Argument(s)**	**Description**
performCustomSetup		Allows the programmer to make adjustments to the starting scene; adjustments that could not be easily made in the Scene Editor. Add program statements to this procedure as is done in any method in Alice. However, all statements here will be executed after the ***Run…*** button is clicked, but before the runtime window is displayed

performGeneratedSetup		When the ***Run…*** button is clicked, Alice inspects the scene built in the Scene Editor and generates the appropriate code necessary to display the scene created by the user in the runtime window. **NOTE: the programmer should not attempt to add or modify code in this procedure, as it is always rewritten whenever the *Run…* button is clicked.**
initializeEventListeners		This procedure of the **Scene** class is the preferred location in an Alice project for the implementation of event listeners. When the ***Run…*** button is clicked, Alice inspects this procedure and generates the appropriate code necessary to implement the listeners for the project. See section below on listener procedures
handleActiveChanged	*isActive, activationCount*	TO BE IMPLEMENTED
myFirstMethod		This is where an Alice animation starts, once the runtime window is displayed. Normally this is the method where the programmer creates program statements that control the overall execution of the animation. (A possible exception is **performCustomSetUp,** as described above).

This *(scene's)* unique properties are shown in Figure A.24.

Figure A.24 Properties methods for Scene class

The *setters* and *getters* of the **Scene** class are used to adjust the sky color, the lighting, and the amount of fog in a scene as an animation program is running, as summarized in Table A.13. These methods are useful for changing the appearance of the scene while the animation is being performed (not for setting up the scene in the Scene editor). For example, to change the scene from a daytime to a nighttime setting, the color of the sky could be made darker and the light in the scene could be decreased.

Table A.13 Properties setters and getters for Scene class

Procedure	**Argument(s)**	**Description**
setAtmosphereColor	*color*	Sets the *color* of the sky in *this* scene
setAmbientLightColor	*color*	Sets the *color* of the primary light source in *this* scene. Think of it as the color of sunlight in an outdoor scene
setFogDensity	*DecimalNumber*	Used to set the *density* of the fog in *this* scene by setting the *density* value in the

		range of values from **0.0** (no fog) to **1.0** (no visibility of objects within the fog).
setFromAboveLightColor	*color*	Sets the *color* of a secondary light source from above in *this* scene
setFromBelowLightColor	*color*	Sets the *color* of a secondary light source from below in *this* scene
Function	**Return Type**	**Description**
getAtmosphereColor	*color*	Returns the *color* of the sky in *this* scene
getAmbientLightColor	*color*	Returns the *color* of the primary light source in *this* scene; think of it as the color of sunlight in an outdoor scene
getFogDensity	*DecimalNumber*	Returns the value of the density of the fog in *this* scene by getting the *density* value with a range of values from **0.0** (no fog) to **1.0** (no visibility of objects within the fog).
getFromAboveLightColor	*color*	Returns the *color* of a secondary light source from above in *this* scene
getFromBelowLightColor	*color*	Returns the *color* of a secondary light source from below in *this* scene

addListener procedures

Listeners are used for creating interactive programs, especially games. **Interactive** means that the user is expected to use the keyboard, mouse, or some other input device to control the actions that occur as the program is running.

A listener is an object that, as a program is running, "listens" for a targeted event and responds to that event when it occurs. For example, a **mouse-click on object** listener will listen for a user to mouse-click on an object in the scene. When the mouse-click on an object occurs, we say the "targeted event has been triggered." When the event is triggered, the listener executes specified instruction statements in response.

In Alice, to create an interactive program, a Listener object must be added to the scene. A listener object is added to the scene by calling an *addListener* procedure, where *Listener* is a targeted event. For example, *addDefaultModeManipulation* creates a listener object that targets a mouse-click on any object in the scene and responds by allowing the user to drag that object around the scene while the animation is running.

Figure A.25 shows a list of addListener procedural methods. Table A.14 summarizes details about the addListener methods, in terms of what event is targeted and how the listener responds.

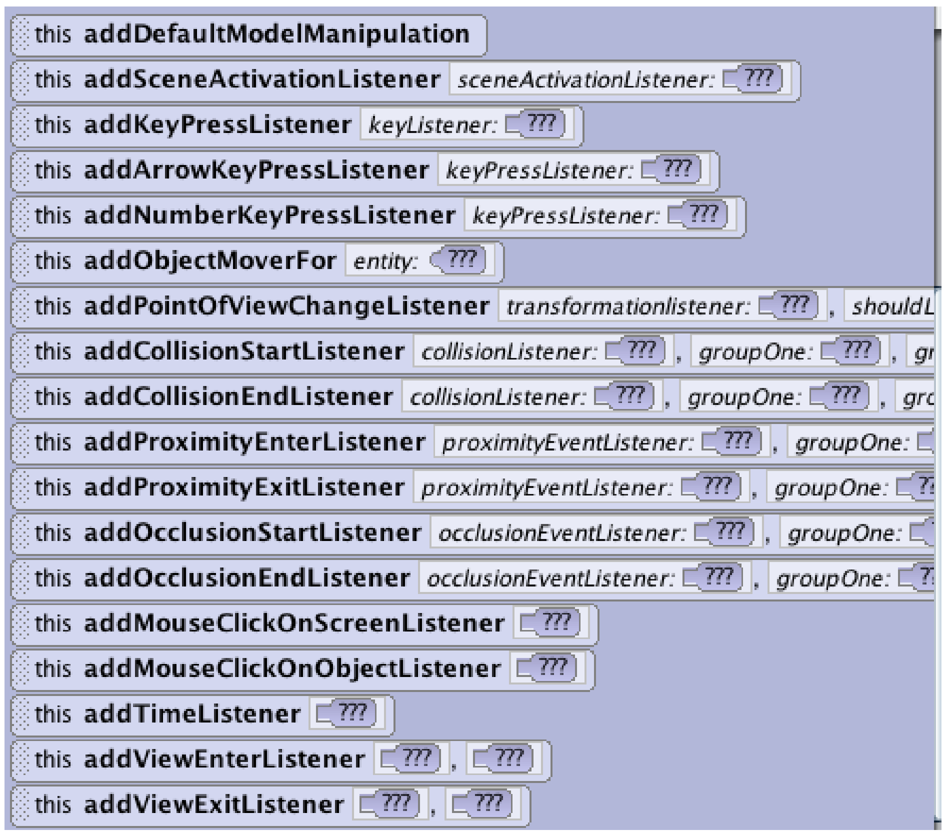


Figure A.25 addListener procedural methods

Table A.14 addListener target and response

Procedure	**Argument(s)**	**Description**
adddefaultModelManipulation		Allows the use the mouse to reposition an object in the virtual world as a

		program is executing. Ctrl-click turns the object, shift-click raises and lowers the object
addSceneActivationListener	*Scene*	UNDER DEVELOPMENT
addKeyPressListener	*Key*	responds to keyboard input from the user. Able to differentiate between Letter, Number, and Arrow keys
addArrowKeyPressListener	*Key*	responds to keyboard input from the user, specifically for Arrow keys (UP, DOWN, LEFT, RIGHT)
addNumberKeyPressListener	*Key*	responds to keyboard input from the user, specifically for Number keys (0..9)
addObjectMoverFor	*Entity*	The parameter object will be moved FORWARD, BACKWARD, LEFT, and RIGHT, based on its own orientation, when the user presses the UP, DOWN, LEFT, and RIGHT arrow keys respectively
addPointOfViewChangeListener	*transformationListener, shouldListenTo*	UNDER DEVELOPMENT
addCollisionStartListener	*collisionListener, Group1, Group2*	UNDER DEVELOPMENT
addCollisionEndListener	*collisionListener, Group1, Group2*	UNDER DEVELOPMENT
addProximityEnterListener	*proximityListener, Group1, Group2, distance*	UNDER DEVELOPMENT
addProximityExitListener	*proximityListener, Group1, Group2,*	UNDER DEVELOPMENT

	distance	
addOcclusionStartListener	*occlusionEventListener, Group1, Group2*	UNDER DEVELOPMENT
addOcclusionEndListener	*occlusionEventListener, Group1, Group2*	UNDER DEVELOPMENT
addMouseClickOnScreenListener	*???*	responds to mouse click input from the user, anywhere on the screen
addMouseClickOnObjectListener	*???*	responds to mouse click input from the user, on the specified object
addTimeListener	*???*	UNDER DEVELOPMENT
addViewEnterListener	???, ???	UNDER DEVELOPMENT
addViewExitListener	*???, ???*	UNDER DEVELOPMENT

Ground

The Ground class has only a limited number of procedural, functional, and property methods, all of which behave exactly the same as those defined by other classes. Figures A.26 (procedures), A.27(functions), and A.28 (specialized property methods) show the methods for the Ground class. These methods were summarized previously in Tables A.4, A.7, and A9.

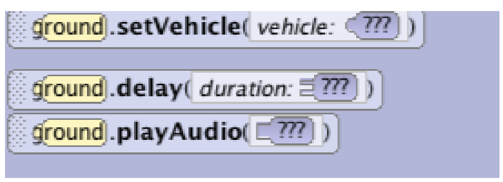


Figure A.26 Procedural methods for Ground class

Figure A.27 Functional method for Ground class

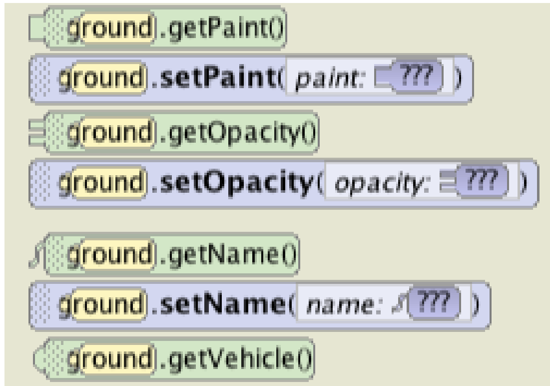


Figure A.28 Specialized property methods for Ground class

Camera

The camera has many procedural methods that behave exactly the same as those defined by other classes, as shown in Figure A.29 and summarized previously in Table A.4.

camera procedures

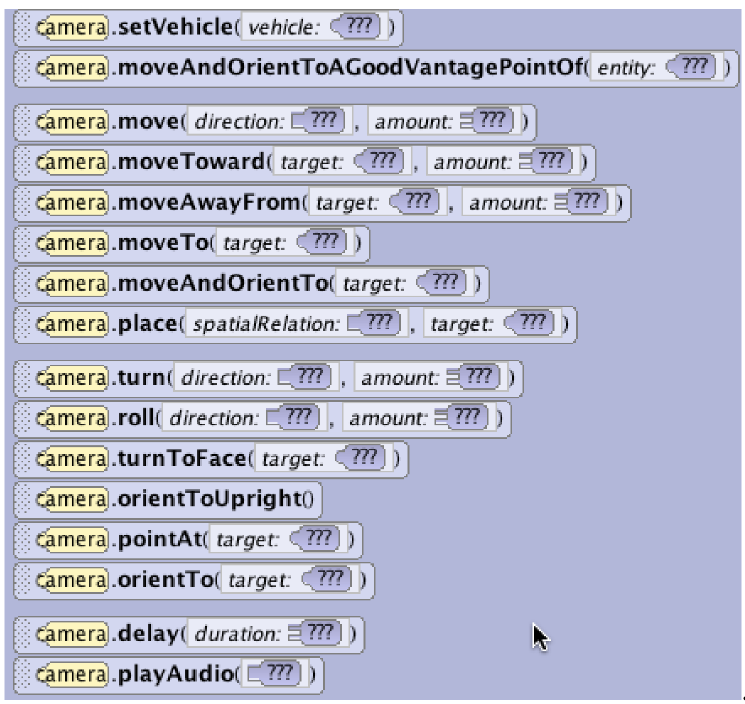


Figure A.29 Camera's procedural methods in common with other classes

One of the procedural methods shown above in Figure A.29, is defined only for the camera: *moveAndOrientToAGoodVantagePointOf*, as described in Table A.15, below.

Table A.15 Unique Procedural method for Camera class

Procedure	Argument(s)	Description
moveAndOrientToAGoodVantagePointOf	*entity*	Animates the reposition and reorientation of the *camera* from its current position to the vantage point of the entity

The Camera class also has only a limited number of functional, and property methods, all of which behave exactly the same as those defined by other classes. Figures A.30 and A.31 show the functional and property methods for the Camera class. These methods were summarized previously in Tables A.7 and A.9.

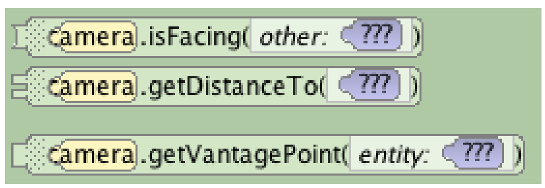


Figure A.30 Camera functional methods

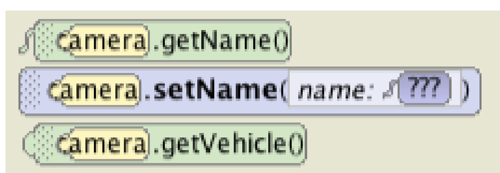


Figure A.31 Camera property methods