

4 Character-Level Methods and Inheritance

The galleries of 3D models (characters) in Alice give us a choice of diverse and well-designed kinds of objects for populating and creating a scenic backdrop in a virtual world. As you know, each character object added to a world comes with a predefined set of actions it can perform – move, turn, roll, and resize (to name a few). After writing several programs, it is natural to think about expanding the actions an object “knows” how to perform.

In fact, we **can** add functionality to any character by writing a method specifically designed for that kind of object to carry out. In object-oriented programming, methods that add functionality to a particular kind of object (a *class* of objects) are called *class-level* methods. In Alice, we use the term *character-level* because we think of a specific kind of object as a 3D character model.

This chapter will focus on character-level methods as methods designed specifically for one kind of object. These methods are rather special because we can save the character along with its newly defined method(s) as a new kind of character. The new kinds of objects still know how to perform all the actions of the original character. In object-oriented languages, we would say that the new kinds of objects (a derived class) *inherit* all the properties and actions of the original objects (the base class). This concept is known as *inheritance*.

Once a new character has been saved, it can then be used later in a different program. Creating new characters in this way offers an advantage in that we can reuse code in many different programs.

Notes to instructors:

1) Creating a new character in Alice is not a complete implementation of inheritance. When a new character model is created in Alice, the new character gets a copy of the properties and methods of the base character model and this is saved in a new 3D model file. If you subsequently change the base character model, the changes in the base will not be reflected in derived character model.

2) Inheritance is accomplished in object-oriented programming languages via two mechanisms: a) adding (or overwriting) methods, and b) adding extra state information via the use of variables. This chapter will focus only on the former approach. Chapter 13 will introduce the use of variables to add functionality to an inherited character. Because variables are not visible/visual in the way the rest of the Alice environment is, variables per se are introduced much later in the text, after students have developed a mastery of several other programming concepts. See the preface for a more detailed discussion of our use of variables in Alice.

4-1 Creating New Characters

In the previous chapter, world-level methods were written to animate more than one object in a scene, where the objects were interacting with one another in some fashion. This chapter will focus on character-level methods. A major difference between world-level methods and character-level methods is that a character level method is specifically designed for one kind of object. The modified character can be saved out to create a new character.

Example

Consider the ice skater shown in the winter scene of Figure 4-1-1. We want the ice skater to perform typical figure skating actions. It would be most convenient if the skater already knew how to skate forward and backward and perform spin and axel movements. But, as with other characters from the gallery, the ice skater only knows how to perform simple move, turn, and roll actions. So, let's write some methods to teach the skater how to perform more complex movements. We begin with a method to make the skater perform a skate forward motion.



Figure 4-1-1. Ice skater

A Character-Level Method

Skating movements are complex actions that require several motion instructions involving various parts of the body. A possible storyboard for a skate forward movement is shown below. The storyboard breaks the skating action up into two segments – slide on the left leg and slide on the right leg. Then, each segment is planned (a design technique known as *stepwise refinement*). To slide on the left leg, the right leg is lifted and body rolled right. Then, the left leg is lowered and the body is made upright. Similar actions are carried out to slide on the right leg. The sliding actions are all taking place at the same time as the entire skater's body is moving forward.

```
Do together
  Do in order
    (Slide Left)
      Lift right leg and roll upper body right
      Lower right leg and return body upright
    (Slide Right)
      Lift left leg and roll upper body left
      Lower left leg and return body upright
  Move forward 2 meters
```

This method is designed specifically for the IceSkater and involves no other kinds of objects. Thus, instead of world-level, the method is written as a character-level method. IceSkater is selected in the Object tree and the “create new method” tile is clicked in the details pane. *skate* is entered as the name of the new method. (See Figure 4-1-2.)

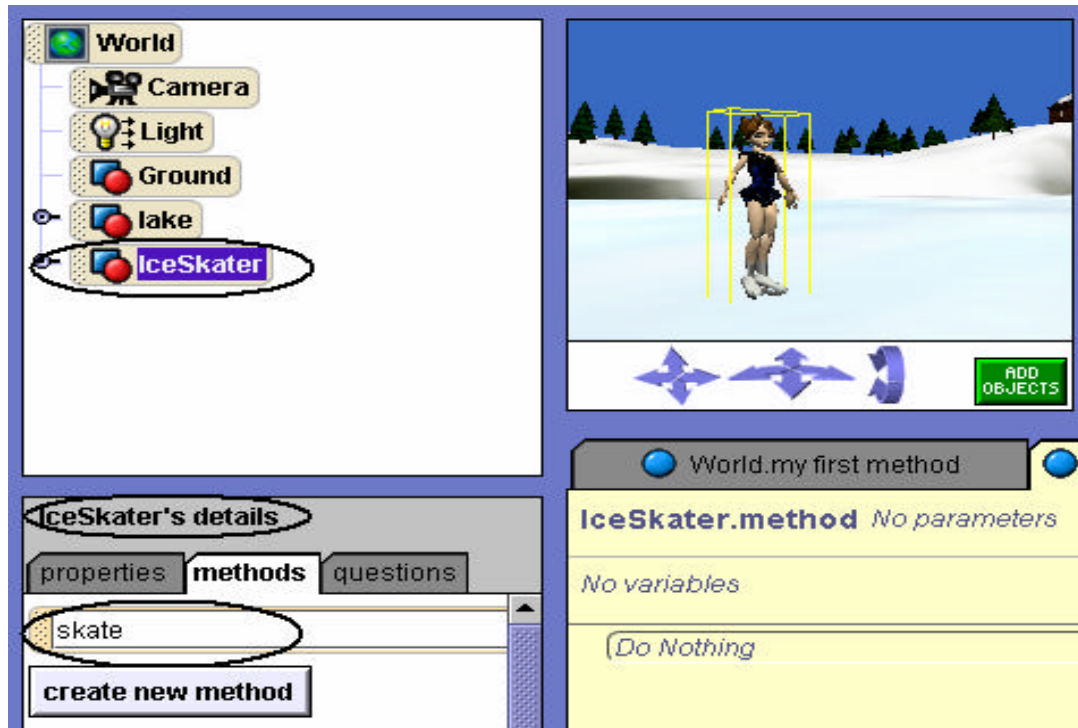


Figure 4-1-2. Creating a *skate* character-level method

To implement the method, instructions are entered in the editor. Figures 4-1-3(a) and (b) make up the definition of the *skate* method. The method name is *IceSkater.skate* (not *World.skate*) – indicating that the method is a character-level method. While the method looks somewhat complicated, you should not be intimidated by this code. It is actually quite simple – a slide left followed by a slide right and the whole body moves forward.

Figure 4-1-3(a) illustrates instructions to slide on the left leg:

The right leg is lifted (by turning the right thigh forward), the upper part of the body (abs) turned forward, and the head tilted (by rolling the neck left). A short wait is used to allow the entire skater to move forward. Then, the right leg is lowered and the abs and neck returned to their original positions (to prepare for the next leg slide).

Figure 4-1-3(b) illustrates similar instructions to slide forward on the right leg.

Note that left slide and right slide segments are each enclosed in a *Do in order* block. The *Do in order* blocks are nested within a *Do together* block along with an instruction that moves the skater forward simultaneously with the left and right sliding motion. The duration of the forward movement of the skater is the **sum of the durations of the left and right slides** so as to coordinate the sliding motions to begin and end at the same time as the forward motion of the entire body. When the *skate* method is called, the skater glides forward in a realistic motion.

World.my first method **IceSkater.skate**

IceSkater.skate No parameters create new param

No variables create new varia

Do together

Do in order

// slide on left leg

Do together

IceSkater.ThighR turn forward 0.1 revolutions duration = 0.5 seconds more...

IceSkater.Abs turn forward 0.01 revolutions duration = 0.5 seconds more...

IceSkater.Abs.Chest.Neck roll left 0.01 revolutions duration = 0.5 seconds more...

Wait 0.5 seconds

Do together

IceSkater.ThighR turn backward 0.1 revolutions duration = 0.5 seconds more...

IceSkater.Abs turn backward 0.01 revolutions duration = 0.5 seconds more...

IceSkater.Abs.Chest.Neck roll right 0.01 revolutions duration = 0.5 seconds more...

Figure 4-1-3(a) Slide left

// slide on right leg

Do together

IceSkater.ThighL turn forward 0.1 revolutions duration = 0.5 seconds more...

IceSkater.Abs turn forward 0.01 revolutions duration = 0.5 seconds more...

IceSkater.Abs.Chest.Neck roll right 0.01 revolutions duration = 0.5 seconds more...

Wait 0.5 seconds

Do together

IceSkater.ThighL turn backward 0.1 revolutions duration = 0.5 seconds more...

IceSkater.Abs turn backward 0.01 revolutions duration = 0.5 seconds more...

IceSkater.Abs.Chest.Neck roll left 0.01 revolutions duration = 0.5 seconds more...

// entire ice skater moves forward as legs slide left then right

IceSkater move forward 2 meters duration = 3 seconds more...

Figure 4-1-3(b) Slide right

A Second Example

The forward skate motion is truly quite impressive! Building on this success, let's write a second method to make the ice skater perform a spin. A storyboard for a spin could be as follows:

Do in order

1. Do together
 - (1) Lift left leg up in preparation for spin
 - (2) Lift arms out to the side
2. Do together
 - (1) Bend leg (at knee) in and out during spin
 - (2) Lower arms gradually as spin occurs
 - (3) Turn skater around six times
3. Lower left leg back to starting position

A character-level method can now be written with the storyboard as a guide. Figures 4-1-4 (a) – (d) make up the definition of the *IceSkater.spin* method.

Figure 4-1-4(a) illustrates a DoTogether block of instructions to prepare the skater for a spin:

The left leg is lifted to the side by turning the left thigh left and then backward. Then, the upper arms are rolled outward – away from the torso of the body.

Figure 4-1-4(b) illustrates a DoTogether block to move the left leg in and out as the body spins around:

The left leg is bent at the knee by first turning the thigh left and then the calf backward. After a short wait, the knee is unbent by reversing the thigh and calf movements.

Figure 4-1-4(c) continues the Do together loop started in Figure 4-1-4(b):

The arms are gradually returned to their starting position as the skater turns around and around.

Figure 4-1-4(d) illustrates instructions to return the skater's left leg to its starting position.

Once again, the code is a bit longer than methods we have written in previous examples. But, it is important to realize that it is easily understood because the method has been carefully broken down into small segments and each small segment of code within the method has been well-documented with comments that tell us what the code segment accomplishes as part of the method. Good design and comments sprinkled throughout the method make our code easier to understand as well as easier to write and debug.

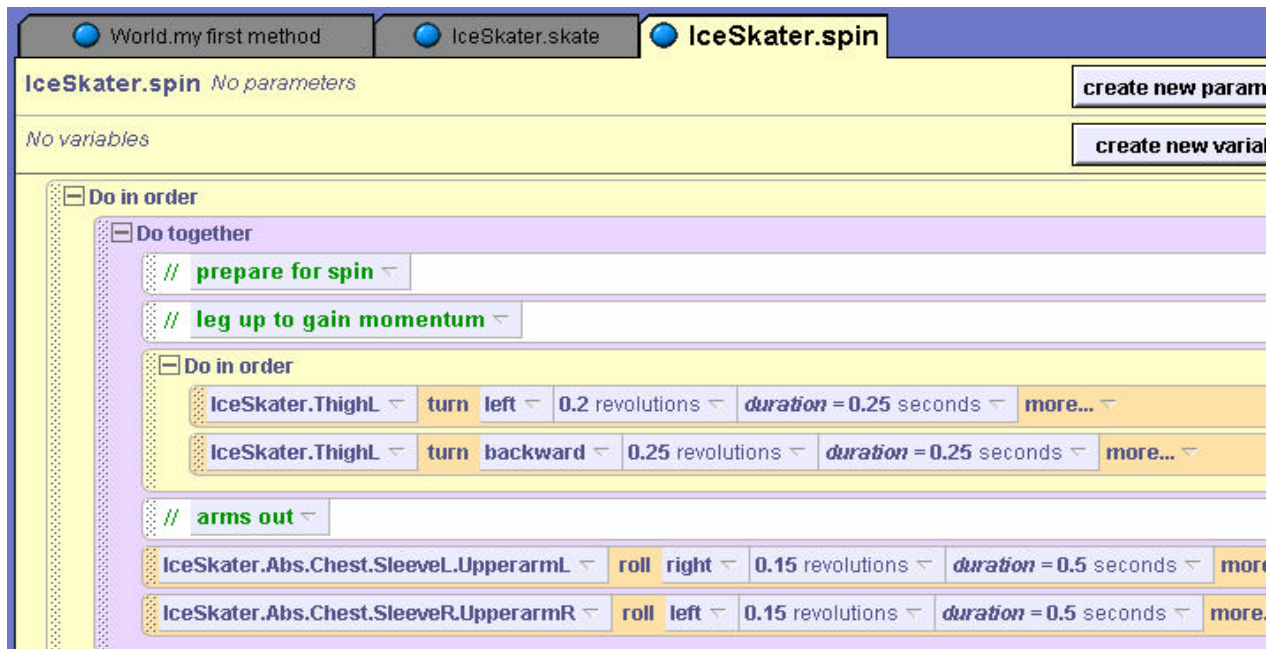


Figure 4-1-4. (a) Preparing for spin

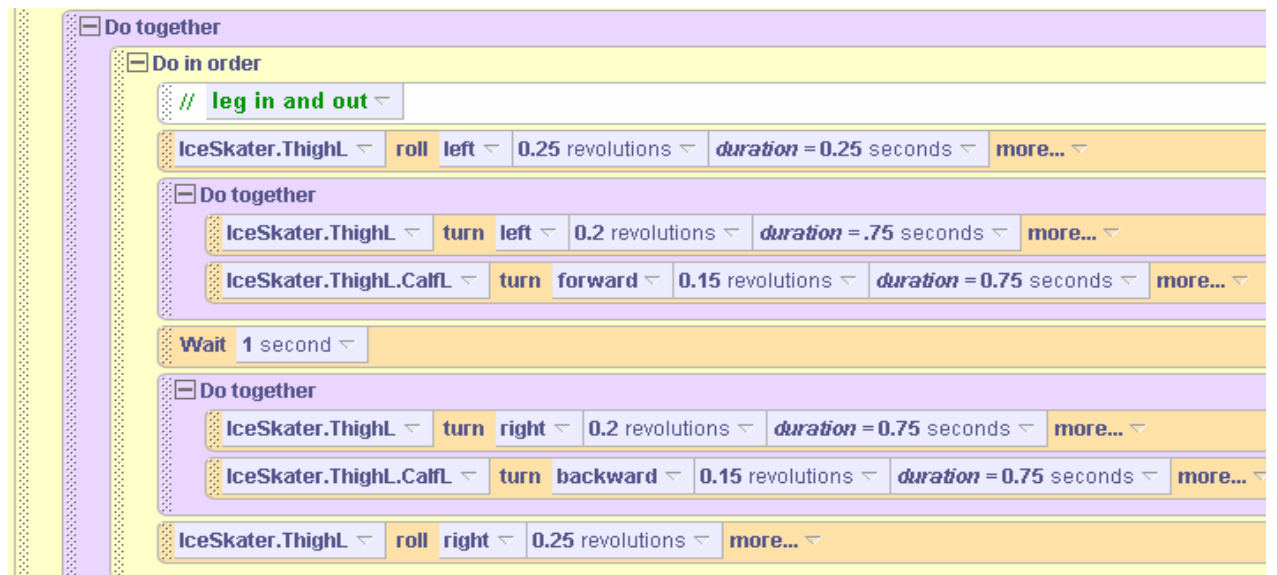


Figure 4-1-4. (b) Leg in and out as skater spins

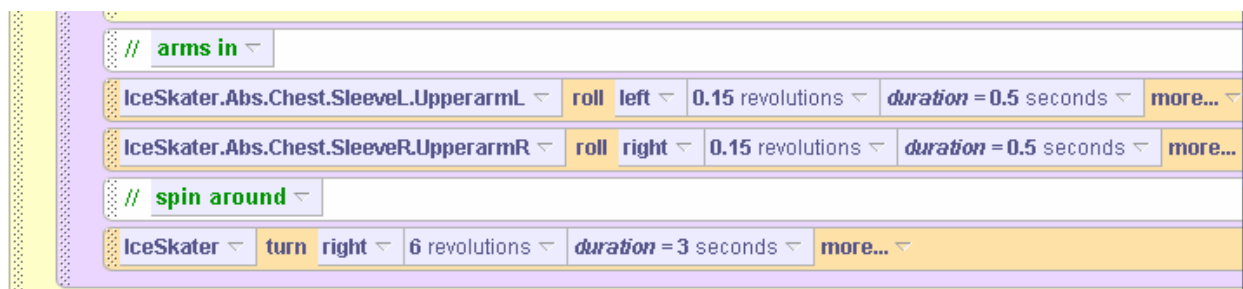


Figure 4-1-4. (c) Arms come in as skater spins

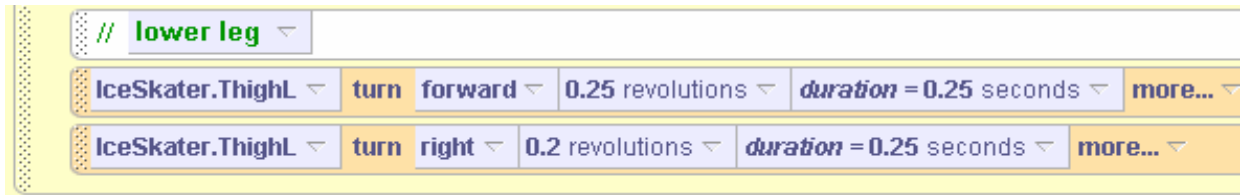


Figure 4-1-4. (d) Return left leg to starting position after spin

Saving a New Character

The IceSkater now has two character-level methods, *skate* and *spin*. Writing and testing the methods took some time and effort to achieve. It would be a shame to put all this work into one world and not be able to use it again in another animation program we might create later. This is why we want to save an object and its newly defined methods as a new character. Saving the IceSkater (with *skate* and *spin* methods) as a new kind of character is exactly what we want to do – so we can use the IceSkater in other worlds and not have to write these methods again in another animation program.

Saving an object (with its newly defined methods) as a new character is a two-step process. The first step is to rename the object. This is an IMPORTANT STEP! We want Alice to save this new character with a different 3D model filename than the original ice skater. To rename the IceSkater, we right click on the IceSkater in the Object tree, and select *rename*. Then, type in a different name. In Figure 4-1-5, we renamed the IceSkater as “CleverSkater”. (She has learned some clever moves.)



Figure 4-1-5. Renaming IceSkater as CleverSkater

The second step is to actually save the CleverSkater: right click on CleverSkater in the Object tree and this time select *save object...* In the Save Object popup box, navigate to the folder/directory where you wish to save the new character, as in Figure 4-1-6, and then click the Save button. The file is automatically named with the name of the object in the Object tree and given a filename extension **.a2c**, which stands for “Alice version **2.0** Character” (just as the **.a2w** extension in a world filename stands for “Alice version 2.0 World”).

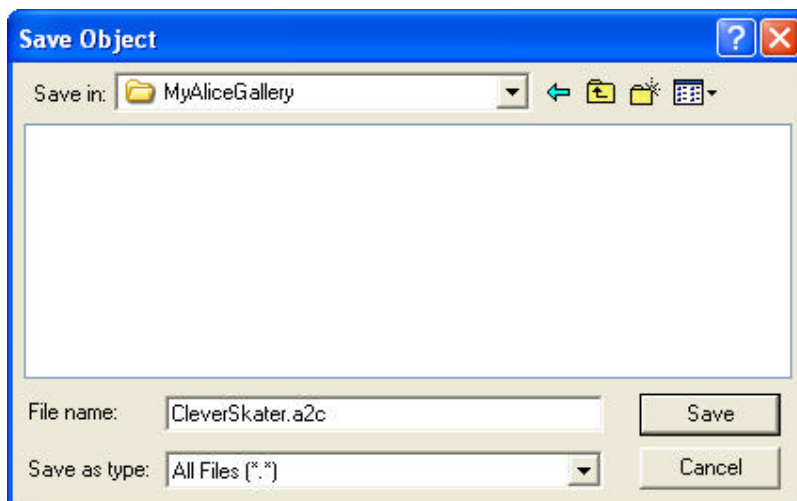


Figure 4-1-6. Save object pop up box

Once the object has been saved as a new kind of 3D model character, it can be used in any world by importing (use File→Import) the object into the scene. The CleverSkater will be just like IceSkater except that a CleverSkater knows how to skate and spin.

Inheritance

In programming languages such as C++, Java, and C#, creating a new class (kind of objects) based on a previously defined class is called *inheritance*. Inheritance in these languages is more complicated than in Alice, as they offer different forms of inheritance. But, the basic idea is the same – adding functionality to the base object by defining behaviors (methods) for a new kind of object.

Benefits: reuse and teamwork

Inheritance is considered one of the strengths of object-oriented languages because it supports reuse of code. Many programmers devote much of their time rewriting code they have written before. But, inheritance allows a programmer to write code once and use it again later in a different program.

A major benefit of creating new characters in Alice is that it allows us to share code with others in team projects. If you are working on a project as a team, each person can write character-level methods for one of the characters in the virtual world. Then, each team member can save their new character. As a team, a common world can then be created by adding all the new characters to a single team-constructed world. This is a benefit we cannot stress enough. In the “real world”, computer scientists generally work on team projects. Building animation programs as a team helps you develop skills in learning to work cooperatively with others.

Guidelines for Writing Character-level Methods

Character-level methods are a powerful feature of Alice. But, as with many powerful features in programming languages, character-level methods have the potential for misuse. Below is a list of “*do's and don'ts*” – guidelines to follow when creating character-level methods.

1. **Do** create many different character-level methods. They are extremely useful and helpful. Some characters in Alice already have a few character-level methods defined. For example, the lion character has methods *startStance*, *walkForward*, *completeWalk*, *roar*, and *charge*. Figure 4-1-7 shows a thumbnail image for the lion character (from the web gallery), including its character-level methods and sounds.

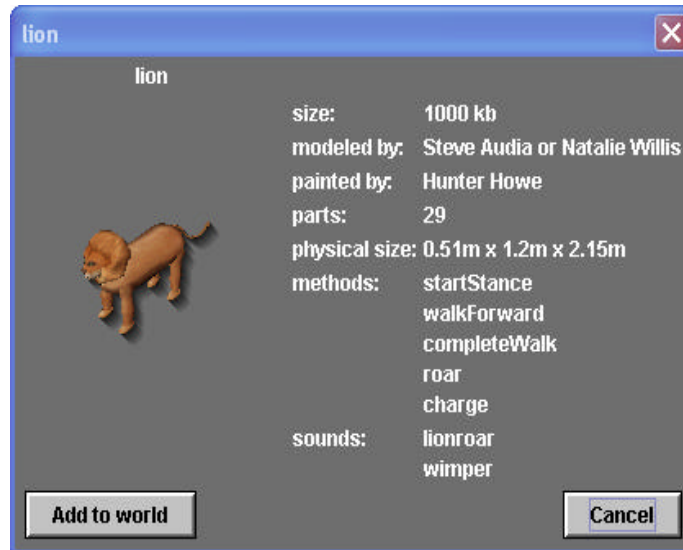


Figure 4-1-7. Character-level methods for lion

2. **Do not** play a sound in a character-level method unless the sound has been imported for the character (instead of the world). A sound that has been imported for a character will be saved with the character when a new character is saved, as can be seen in Figure 4-1-7.
3. **Do not** call world-level methods from within a character-level method. Figure 4-1-8 illustrates *CleverSkater.kalidoscope* -- a character-level method that calls a world-level method named *World.changeColors*. As explained earlier, an important motivation for creating character-level methods is to create a new character (saved as a new 3D model) and use it later in other programs for other worlds. If the *CleverSkater* object (with the *CleverSkater.kalidoscope* method) is saved as a new character and is then added to a later world where the *World.changeColors* method has not been defined, Alice will crash.

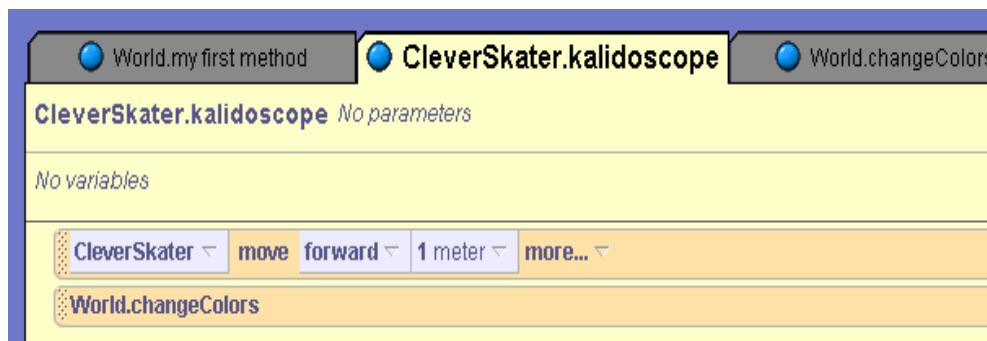


Figure 4-1-8. **Bad Example:** Calling a world-level from a character-level method

4. **Do not** use instructions for other objects from within a character-level method. Character-level methods are clearly defined for a specific character. We expect to save the object as a new kind of character and reuse it in a later world. We cannot depend on other objects being present in other programs in other worlds. For example, suppose a Penguin is added to our winter scene and we define a character-level method named *skateAround*, as in Figure 4-1-9. If we save the CleverSkater object (with the skateAround method) as a new kind of character and then add a CleverSkater object to a later world where no penguin exists, Alice will crash when we try to call the method.

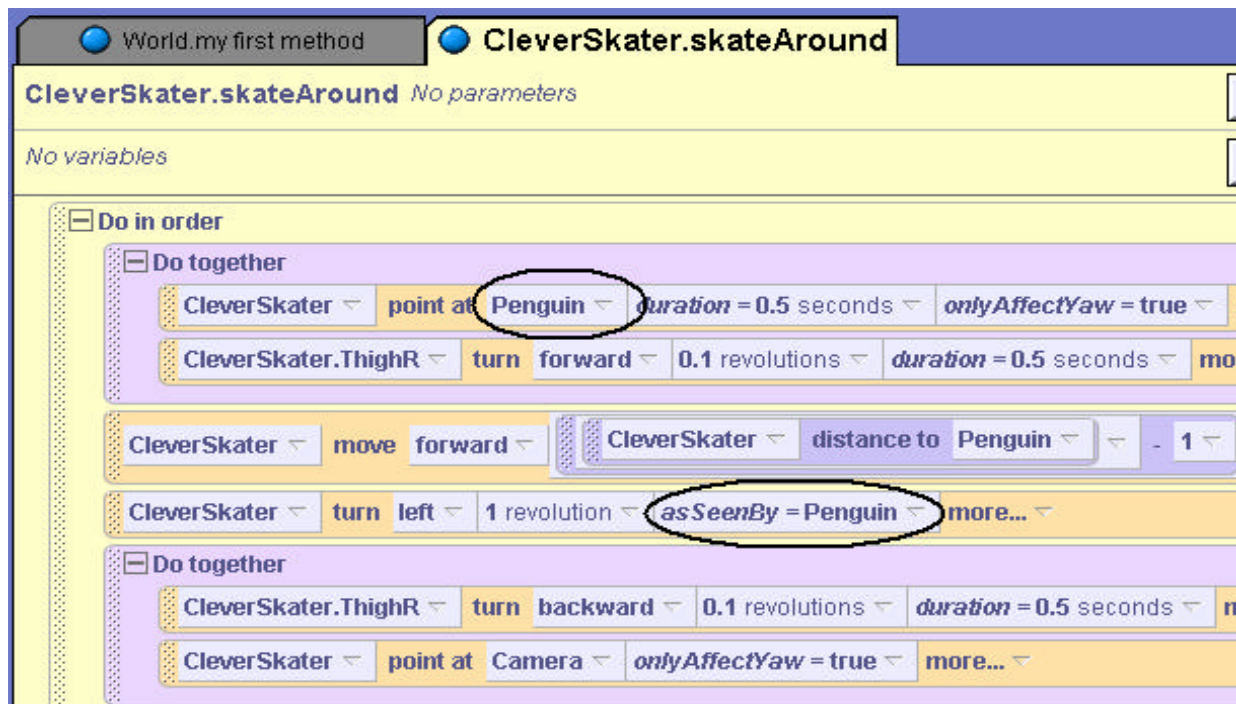


Figure 4-1-9. **Bad Example:** Instructions for another object in a character-level method

Character-level Method With an Object Parameter

What if you are **absolutely convinced** that a character level method is needed where another object is involved? One technique is to use an object parameter in the character-level method. Let's use the same example as above, where we want a CleverSkater object to be able to skate around another object. The *skateAround* method can be modified to use a parameter, arbitrarily

named *whichObject*, as shown in Figure 4-1-10. The *whichObject* parameter is only a placeholder, not an actual object, so we do not have worry about a particular object (like the penguin) having to be in another world. Alice will not allow the *skateAround* method to be invoked without passing in an object to the *whichObject* parameter. So, we can be assured that some sort of object will be there to skate around.

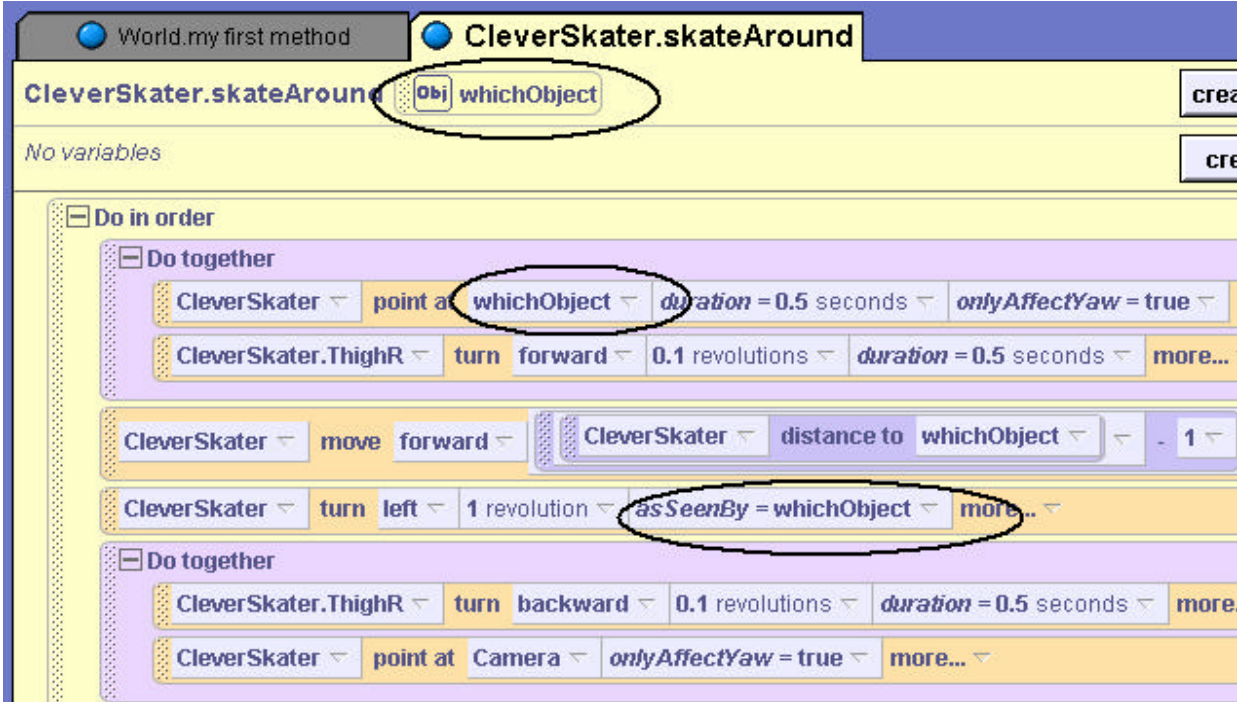


Figure 4-1-10. Using an object parameter in a character-level method

4-1 Exercises

1. LockCombination

Create a world with a ComboLock. Create four character-level methods: *leftOne*, *rightOne*, *leftRevolution*, and *rightRevolution* that turn the dial on the lock 1 number to the left, 1 number to the right, 1 revolution to the left, and 1 revolution to the right, respectively. Then, create a character-level method named *Open* that opens the lock and another named *Close* that closes the lock. (Hint: Use an *end Gently* style to make the motion more realistic.) Rename ComboLock as **TurningComboLock** and save it as a new kind of character.



Create a new world and add a TurningComboLock object to the world. Write a program that turns the dial of the TurningComboLock object so it opens to the following combination: Left 25, Right 16, Left 3. However, when going Right 16, make sure that the dial turns one full revolution by passing 16 once and stopping on it the second time. Then, pop open the latch, close the latch and return the dial to zero. (*Hint: Use wait to make the lock pause between each turn of the dial.*)

2. FunkyChicken

Starting with a basic chicken, create a character-level method *walk* that will have the chicken perform a realistic stepping motion consisting of a left step followed by a right step. Then, create a character-level method to make the chicken perform a “*funkyChicken*” dance motion. Save the chicken as a new character named **CoolChicken**. Create a new world and add a CoolChicken object to the world. In *myfirstMethod*, call the *walk* and *funkyChicken* methods. Play a sound file or use a *say* instruction to accompany the funky chicken dance animation.



3. Samurai Practice

Create a world with a Samurai and write character methods for traditional Samurai moves. For example, you can write *RightJab* and *LeftJab* (where the Samurai jabs his hand upward with the appropriate Hand), *KickLeft* and *KickRight* (where he kicks with the appropriate leg), and *LeftSpin* and *RightSpin* (where he does a spin in the appropriate direction). Save the Samurai as a new character named **TrainedSamurai**. Start a new world and add two TrainedSamurai objects. Create an animation where the two TrainedSamurai objects practice their moves, facing one another.



4 Summary

Character-level methods were introduced as a special kind of method written specifically for one type of object. An object for which character-level methods are defined may be saved as a new type of object – using a different name than the base character model. The new character inherits the properties and actions of the original character -- it is a fancy new model that knows how to do more things than the base model. Comments and stepwise refinement were used to make complex actions in character-level methods easy to understand and debug.

Some guidelines must be imposed for writing character-level methods. Only sounds imported for the character should be played, world-level methods should not be invoked, and instructions involving other objects should not be used.

A major benefit of defining new characters is we can use the characters over and over again in new worlds to take advantage of the methods we have written without having to write them again.

Important concepts in this chapter

- A character-level method is defined for a specific type of object.
- New character models can be created by defining character-level methods for an object and then saving the character object with a new name.
- Stepwise refinement is a design technique where a complex action is broken down into segments and then each segment is designed and implemented.
- Inheritance is an object-oriented concept where new kinds of objects (derived class) are defined based on an existing kind of object (base class). In Alice, the existing 3D models are like base classes and our new characters are similar to derived classes.
- Character-level methods can be written that accept object parameters. This allows a character-level method to interact with another character. Otherwise, character-level methods should avoid interaction with other characters.

4 Projects

1. CleverSkater

Create an even better CleverSkater than the one presented in this chapter. Start with the IceSkater character from the People collection, create the character-level *skateForward*, *spin*, and *skateAround* methods. In addition, create *skateBackward* and *jump* character-level methods. In *skateBackward*, the skater should perform similar actions to those in the *skateForward* method, but slide backward instead of forward. In a jump, the skater should move backward, bend and lift one leg, then move upward (in the air) and spin around twice before gracefully landing on the ice and lowering her leg back to its starting position.

Start a new world with a winter scene. Add your CleverSkater character to the world and create an animation where the skater shows off her figure skating skills. (Call each of the methods you have written.) Add a penguin and a duck to the world and use a parameter to pass the object the skater is to skate around to the *skateAround* method.

2. Your Own Creation

Choose an animal or a person from one of the galleries. The character selected must have at least two legs, arms, and/or wings that can move, turn, and roll. Write three character-level methods for the object that substantially add to what this kind of object knows how to do. Save the new character with a different name and then add it to a new world. Write an animation program to demonstrate the methods you defined for this kind of object.

Tips & Techniques 4

Properties

Objects have properties where information (about the object) is kept. At the lower left of the window in Figure T-4-1, the lilfish object properties are listed. Properties of objects include color, opacity, vehicle, skin texture, and other values that tell Alice how to display and animate the object.

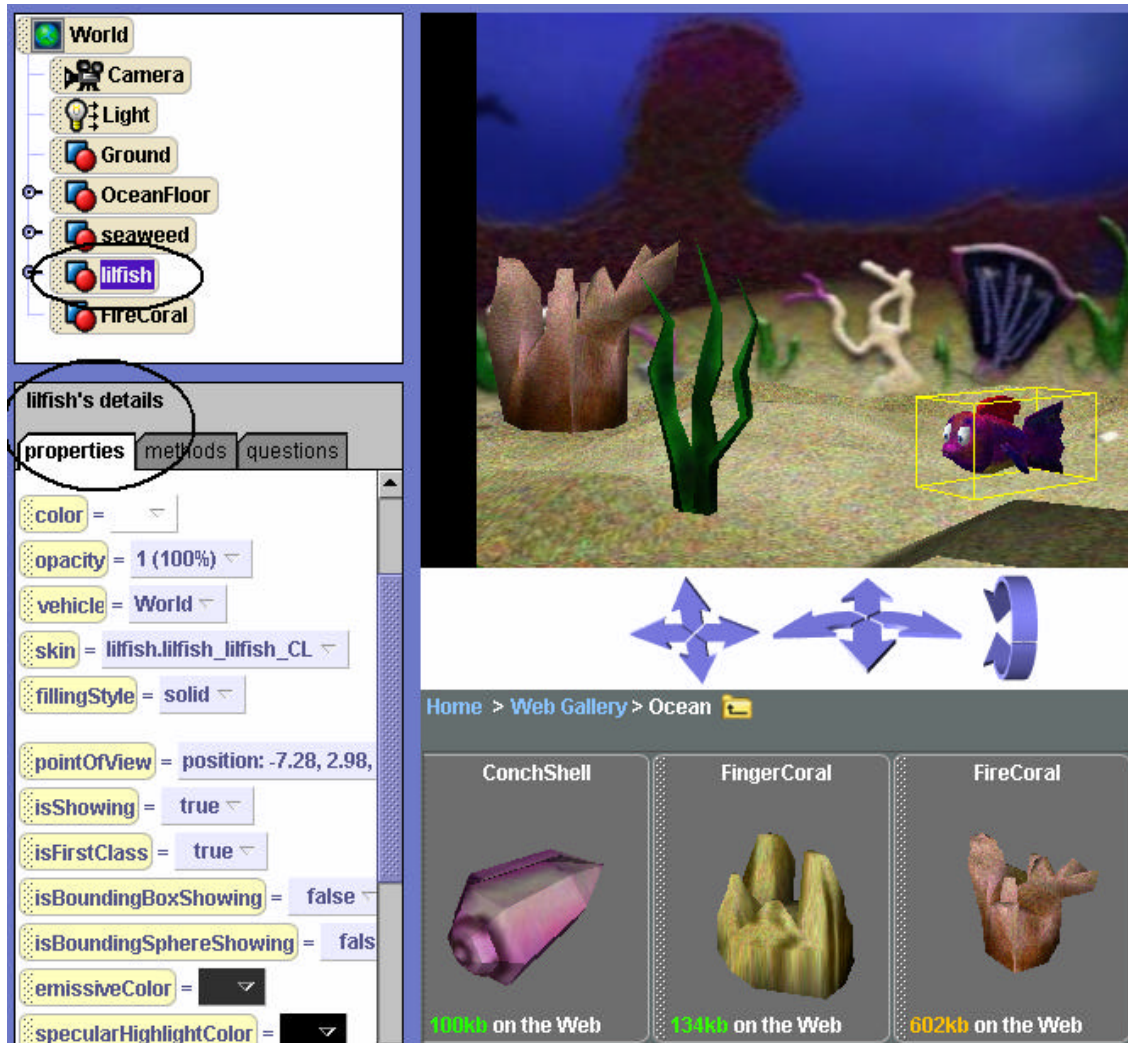


Figure T-4-1. Object properties

Setting Properties at Runtime

Throughout this book, examples use one-shot instructions to set (change) properties of an object at the time an initial scene is created. This section presents a tip & technique on how to change properties while an animation is running. The technical term for “while an animation is running” is “at runtime.”

If you buy a new cell phone, you will take it out of the box and try out all the cool features of your new phone – sort of a “poke and prod” kind of procedure. In the same way, you can learn about properties of objects by taking a “poke and prod” point of view. To set a property at runtime, drag the property tile into the editor and select the new value for the property. Alice will automatically generate a statement in the editor to set the property to the selected value. Set up a world with a couple of objects, drag some property instructions into the editor, and then run the program to see what happens! **Keep trying – on lots and lots of properties until you figure out what they all do.** To get you started, two property change examples are demonstrated here. Other property change instructions are scattered through the remainder of the book, where examples set various properties at runtime.

Setting *Opacity*. Suppose lilfish, as in Figure T-4-1, is swimming out to lunch and her favorite seafood is seaweed. Instructions to point lilfish at the seaweed and then swim toward it are shown below. The *wiggleTail* method is defined to make the fish waggle its tail in a left-right motion. The world and the swim method can be found on the CD with this book.



As the fish moves toward the seaweed, she will also be moving away from the camera. To simulate underwater conditions, we must consider that as lilfish swims away from the camera she should fade because water blurs our vision of distant objects. We can make lilfish become less visible by changing the opacity property. As opacity is decreased, an object becomes less distinct (faded). To change opacity, click on the opacity tile in the properties pane and drag it into the editor. From the popup menu, select the opacity percentage, as shown in Figure T-4-2.

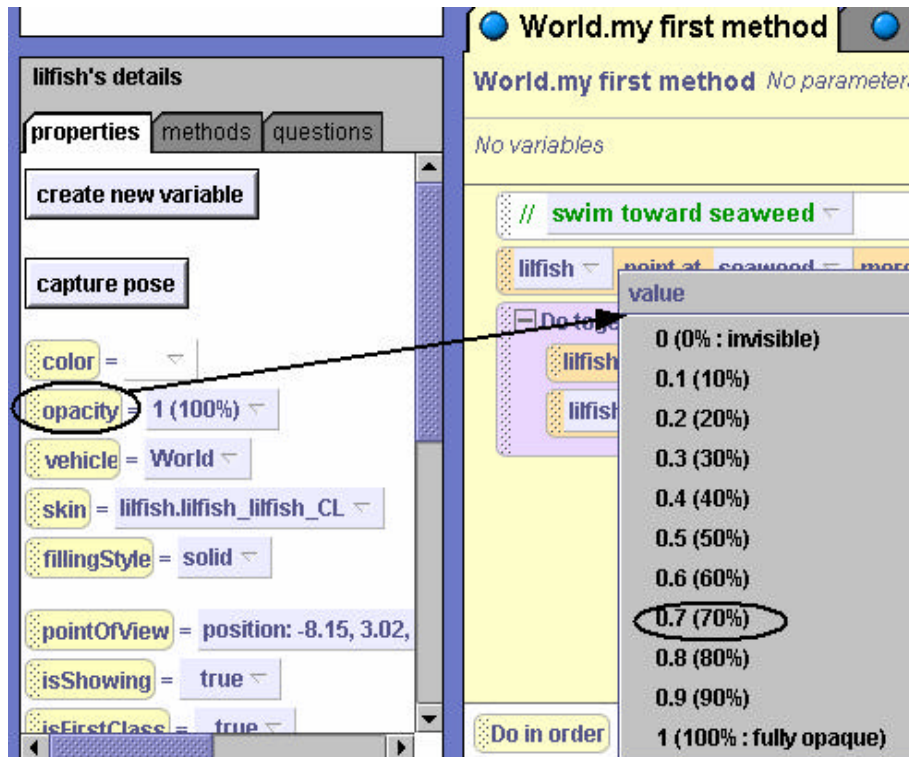
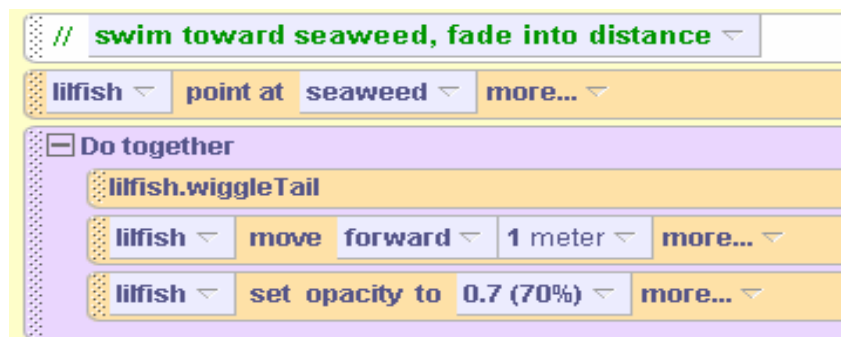


Figure T-4-2. Dragging the *opacity* property into editor

The resulting code looks like this:



When the world is run, the *lilfish* object will fade as shown in Figure T-4-3. Lowering opacity causes the object to become less opaque, and hence less visible. At 0%, the object will totally disappear from view. This does not mean that the object has been deleted. The object is still part of the world but is not visible on the screen.



Figure T-4-3. Fish image fades as opacity is decreased

Setting *isShowing*. Instead of gradually fading away into the distance, suppose we want an object to suddenly disappear? Alice provides a second mechanism for making an object invisible, a property called *isShowing*. At the time an object is first added to a new world, Alice automatically makes the object visible in the scene and *isShowing* is considered *true*. When we want an object to just evaporate “into thin air,” the *isShowing* property is set to *false* while the animation is running. To set the *isShowing* property to false, drag the *isShowing* property tile into the world and select false from the popup menu, as shown in Figure T-4-4.

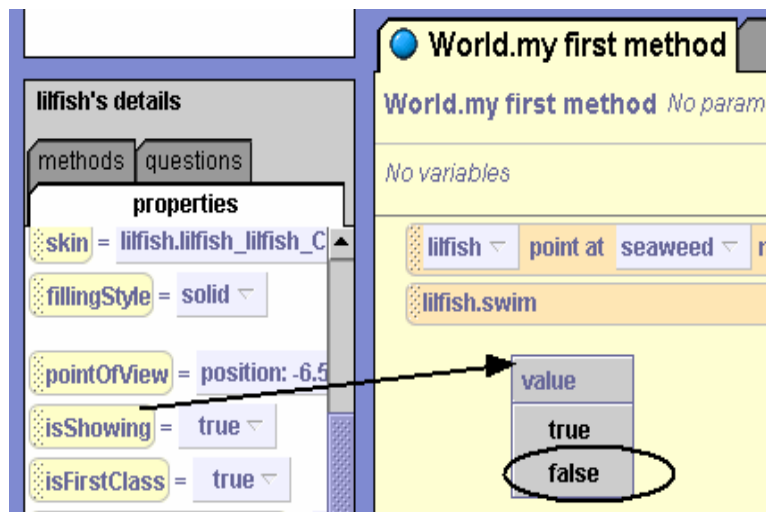
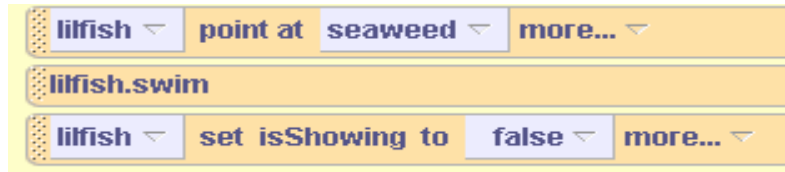


Figure T-4-4. Dragging *isShowing* property into the editor

The resulting code is shown below.



When an object's `isShowing` property is set to `false`, it is not removed from the world. Instead, the object is simply not rendered to the screen. Later in the program, the object can be made to reappear by setting its `isShowing` property back to `true`.

Built-in Questions (Functions)

The Alice system provides a set of built-in *questions* -- statements you can ask about properties of objects and relationships of objects to one another. In other computer programming languages, questions are often called *functions* and a function is said to "return a value."

What kinds of values can you expect to receive when you ask a question in Alice? Values can be any one of several different types. Five of the more common are:

- number (for example, 5)
- logic value (*true* or *false*)
- string (for example, "hello world")
- object (for example, a Robot)
- position (translational and rotational orientation)

What questions can you ask? As with methods, some questions are character-level and some are world-level. Character-level questions are about properties of a specific object in the world such as its height, width, and depth. World-level questions are more utilitarian, having to do with things like the mouse, duration times, and some math operations.

As an example of character-level questions, let's continue with the `lilfish` example used above. To view a list of questions about the `lilfish` object, select `lilfish` in the Object Tree and then the questions tab in the details pane, as in Figure T-4-5.



Figure T-4-4. Object questions

Character-level questions are divided into subcategories:

- *Proximity* – how close the object is (such as distance to, distance above) to some other object in the world
- *Size* – dimensions such as height, width, and depth, and how these compare to the dimensions of another object in the world
- *Spatial relation* – orientation compared to another object in the world (such as to left of, to right of)
- *Point of view* – position in the world
- *Other* – miscellaneous items such as the name of a subpart of the object

The questions shown in Figure T-4-4 are *proximity* questions. Some proximity questions such as “lilfish is within threshold of another object” and “lilfish is at least threshold away from another object” (*threshold* is a distance in meters) return a *true* or *false* value. Other questions such as “lilfish distance to” and “lilfish distance to the left of” return a number which is the distance in meters.

Take on that “poke and prod” attitude to explore all these different kinds of questions in the same way as you explored properties, suggested above. To show you how to get started, here is an example.

Asking about *Distance to*. Life is never easy in the ocean world. You may have noticed that a predator named *uglyfish* has entered the aquatic scene in Figure T-4-4. Of course, **lilfish is not dumb** – she is going to swim to the coral reef (FireCoral object) to hide. The program code needs to be modified to make lilfish swim to the coral. But, how far away is the coral? One way to find out is to use trial and error – that is, try different distances until we find one that works. Another technique to find the distance is to ask Alice a question: “What is the distance of lilfish to the FireCoral?” Alice will answer the question by returning the distance. Then, lilfish can be moved forward that distance. First, modify the code to point lilfish at the FireCoral object. Then, the move forward instruction should be modified to make lilfish swim the distance to the FireCoral object (instead of 1 meter, as previously written). To ask the *distance to* question, first drag the *distance to* tile into the editor and drop it to replace the 1 meter distance. From the popup menu, select the target FireCoral object, as demonstrated in Figure T-4-5.

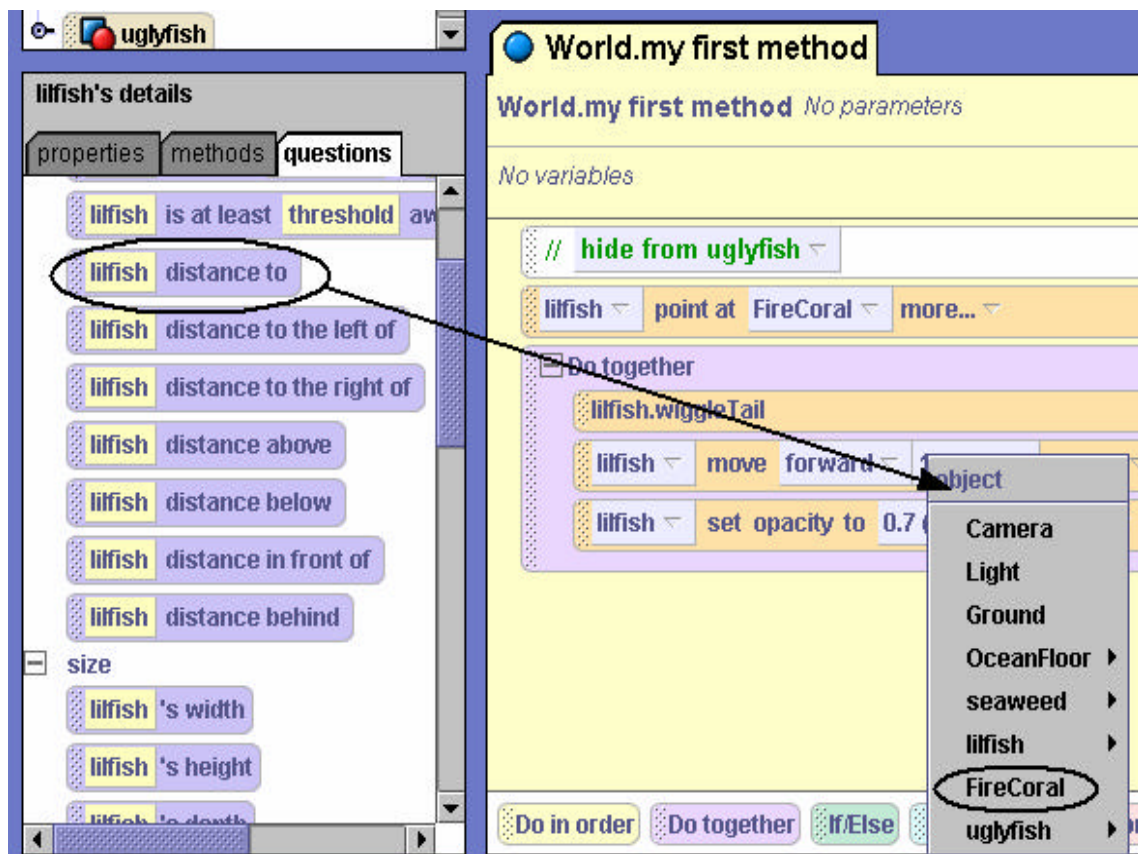
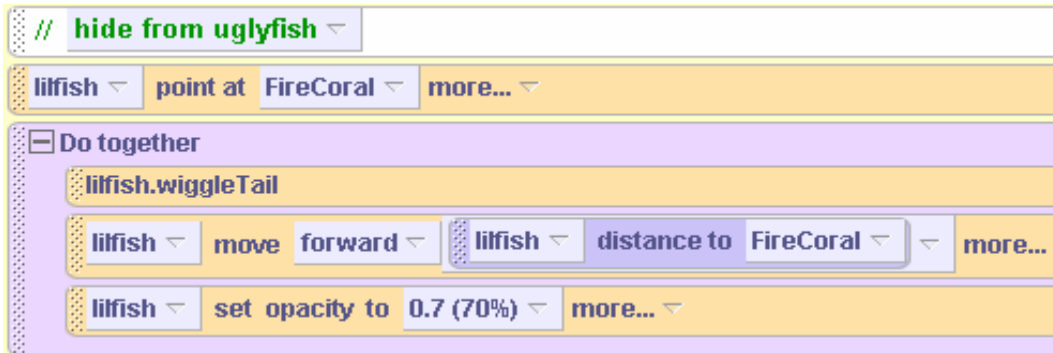


Figure T-4-5. Dragging distance to question into editor

The resulting code is:



Notice that no question mark appears at the end of the call to the question. Alice will interpret this as a question anyway and reply with the distance between the two objects.

Collision


When the program code to hide lilfish from uglyfish is executed, the animation is not exactly what was expected. Lilfish seems to run right into the middle of the FireCoral object. This is called a *collision*. In some animations, a collision is exactly what is desired. But, in this example, **we are not willing to believe that lilfish can swim right through the bony-like crust of the coral object**. The reason a collision occurs is that Alice answers the *distance to* question with a number representing the distance from the center of the fish to the center of the coral. To see how this works, look at Figure T-4-6.



Figure T-4-6. *distance to* is center-to-center

Expressions

How can a collision be avoided between these two objects? One way is to adjust the distance that lilfish moves so she doesn't swim right inside the coral. Adjusting the distance requires the

use of an arithmetic expression. Alice provides arithmetic operators for common arithmetic expressions: add (+), subtract (-), multiply (*), and divide (/). To use an arithmetic expression to adjust the distance the object moves, click  to the right of the distance tile, then select math \rightarrow lilfish distance to FireCoral $-$ \rightarrow 1.25.

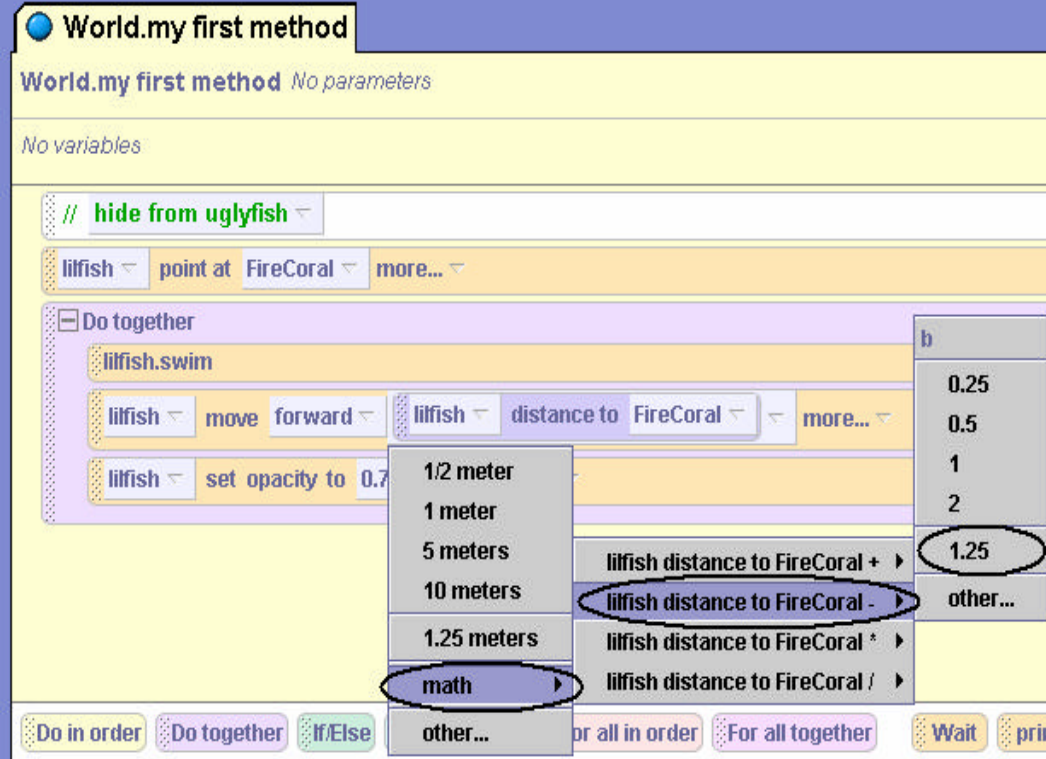
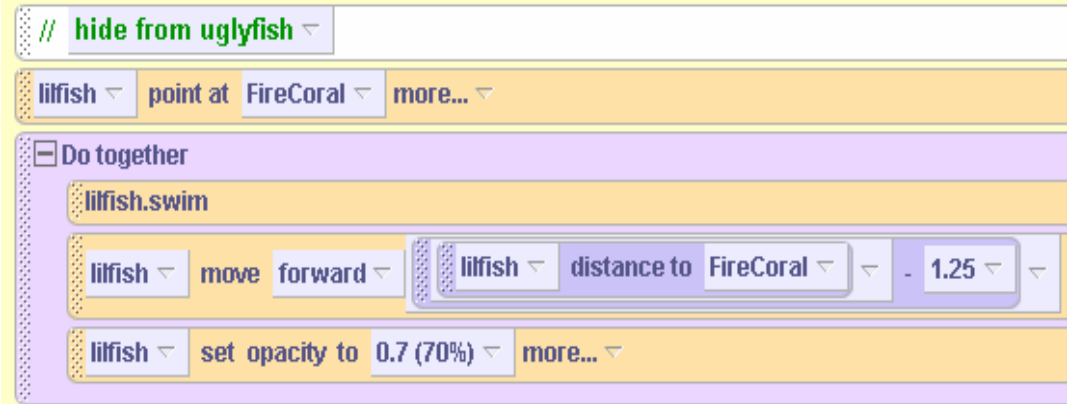


Figure T-4-7. Selecting an expression to adjust distance

The resulting instruction subtracts 1.25 meters from the distance as shown below. Now, the lilfish object will stop short of colliding with the coral.



The move to instruction

A *move to* instruction moves an object to a specific location in a world. The objects in our worlds are, of course, in 3D space. So, a location is specified using a reference to the position of the object along a 3D axis, as shown in Figure T-4-8.

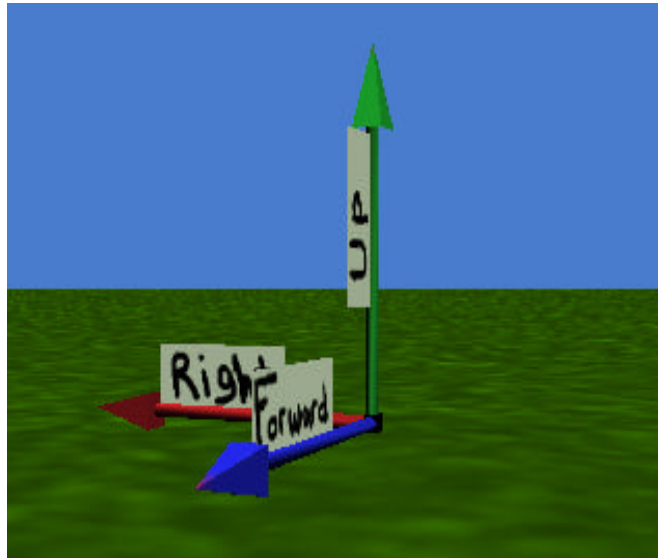


Figure T-4-8. Location reference along 3D axis

When an object is first added to the world, it is automatically positioned at location (0,0,0) which is at the center of the world. The first coordinate specifies the left/right position, the second an up/down position, and the third specifies a forward/backward position relative to the center of the world. Although we can tell Alice to move an object to a location by entering a location using number coordinates, much of the time we use the move to instruction with the position of another object as the target location. (Alice knows the 3-dimensional location of every object in the scene and can move another object to the same location.) This is easier to understand if we show you an example. Figure T-4-9 shows a coach practicing basketball in the gym.



Figure T-4-9. Basketball initial scene

The coach is going to shoot the basketball toward the hoop. We want to animate the movement of the basketball to the rim of the hoop on the backboard. This can be done using a move to instruction with the position of the hoop rim as the target location. Creating the instruction is a two step process.

First, select the basketball in the Object tree. From the basketball's methods, drag the move to instruction into the editor, as shown in Figure T-4-10.

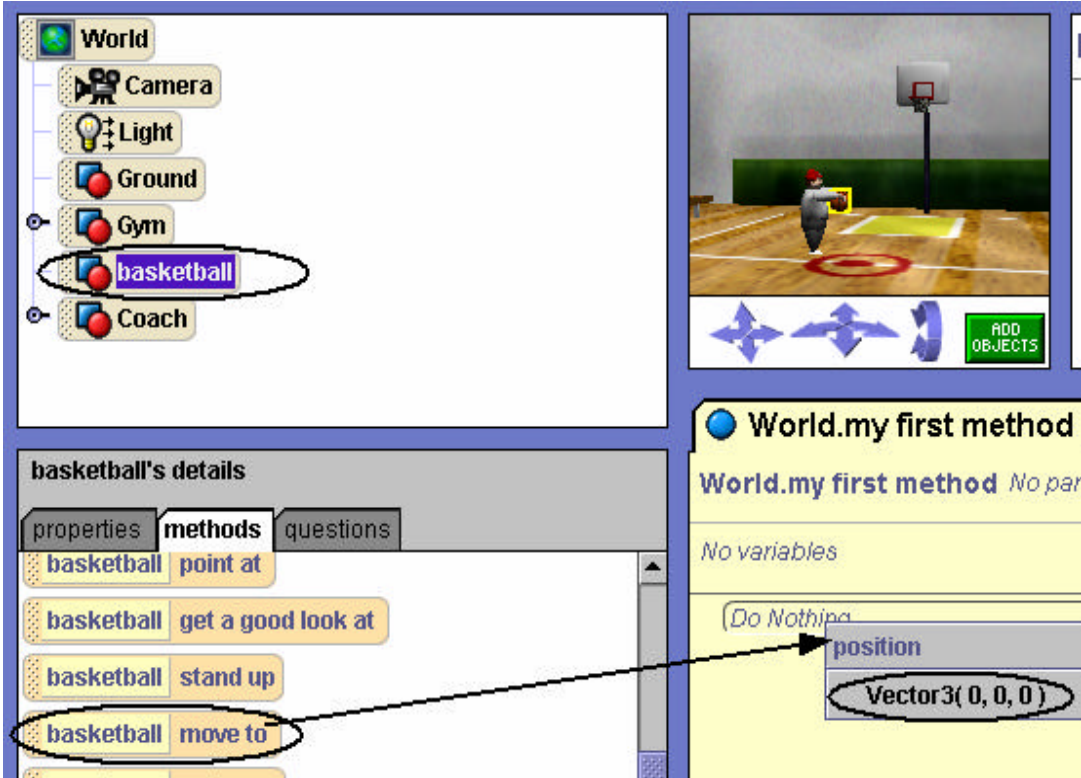


Figure T-4-10. Dragging *move to* into editor

Clearly, when the *move to* instruction is dragged into editor (see Figure T-4-10), the only choice is the default location Vector3 (0,0,0) -- the center of the world. The resulting instruction is:



Step two is to replace the default location with the position of the target object, in this case the hoop rim. To get the position of the hoop rim, drag in the built-in position question for the rim object that asks Alice to return the position of the rim, as shown in Figure T-4-11.

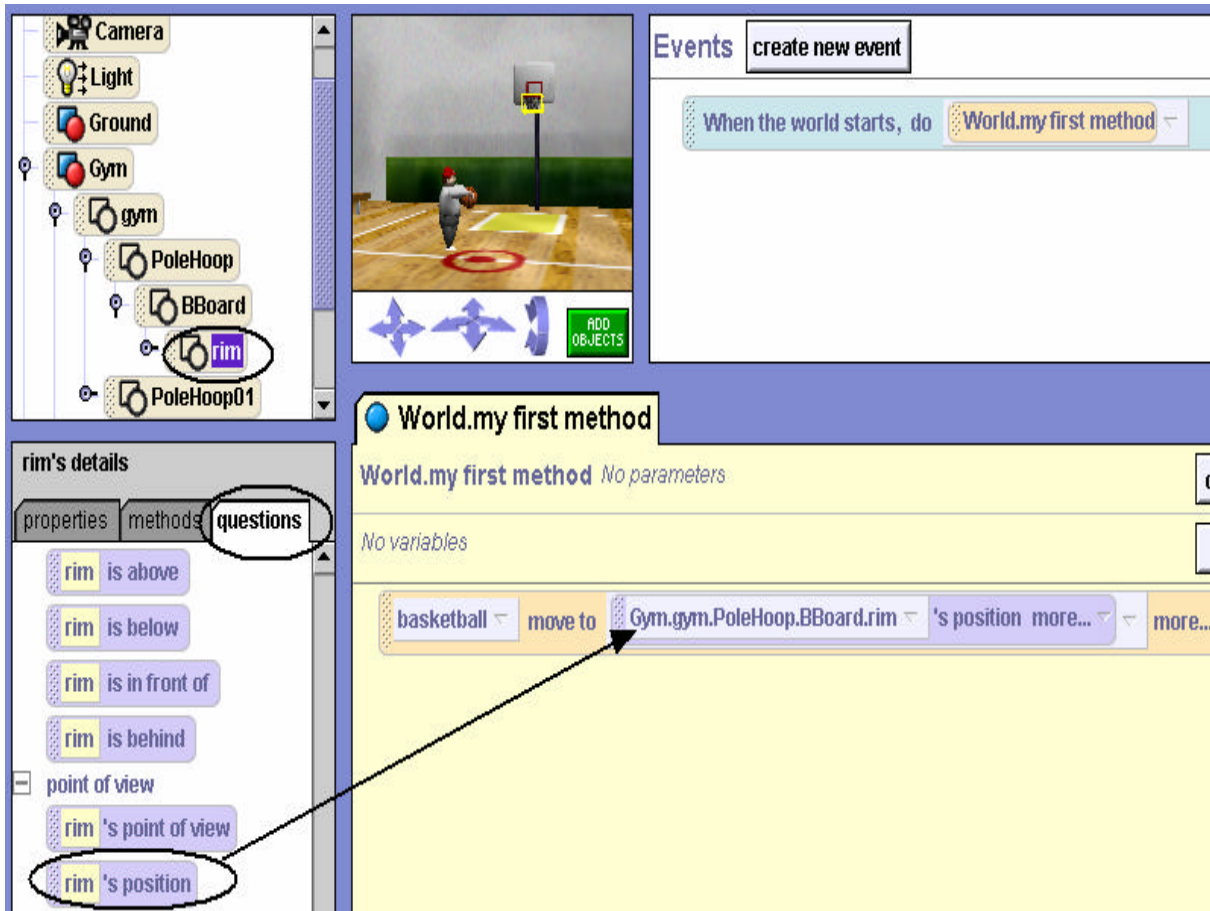


Figure T-4-11. Dragging in the position built-in question

Now, when the instruction is executed, the basketball will move to the rim of the hoop, see Figure T-4-12.



Figure T-4-12. Ball moves to rim

5 Interactive Programs: Events and Event-Handling

The real world around us is interactive. We drive cars that turn right or left when we turn the steering wheel. We change the channel on our television set by sending a signal from a remote control. We press a button on a game-controller to make a character in a video game jump out of the way of danger. It's time we looked at how to create interactive programs in Alice – where the objects in the scenes respond to mouse clicks and key presses. We have concentrated on writing programs that were non-interactive – we watched the objects perform actions in a movie-style animation. In this chapter, we will see how programs can be made interactive.

Much of computer programming (and of the Alice movie-style animations seen earlier) is computer-centric. That is, the computer program basically runs as the programmer has intended it. The programmer sets the order of actions and controls the program flow. However, many computer programs today are user-centric. In other words, it is the computer user (rather than the programmer) who determines the order of actions. The user clicks the mouse or presses a key on the keyboard to send a signal to Alice about what to do next. The mouse click or key press is an *event*. An *event* is something that happens. In response to an event, an action (or many actions) is carried out. We say the “*event triggers a response.*”

Section 5-1 focuses on the mechanics of how the user creates an event and how the program responds to the event. Naturally, all of this takes a bit of planning and arrangement. We need to tell Alice to listen for a particular kind of event and then we need to tell Alice what to do when the event happens. This means we need to write methods that describe the actions objects in the animation should take in response to an event. Such a method is called an *event-handler* method.

Section 5-2 describes how to pass parameters to *event-handler* methods. In some programming languages, arranging events and writing event-handler methods is a rather complex kind of programming. But, one of the achievements of Alice is that the event-response model is sufficiently simple to illustrate this material to novice programmers.

A special note to instructors:

We have found that interactive programs are fun and highly motivating to students. Nonetheless, this chapter may be safely skipped from a pedagogic perspective. Almost all exercises and projects in this book can be created in a non-interactive style.