Programming

in

Alice

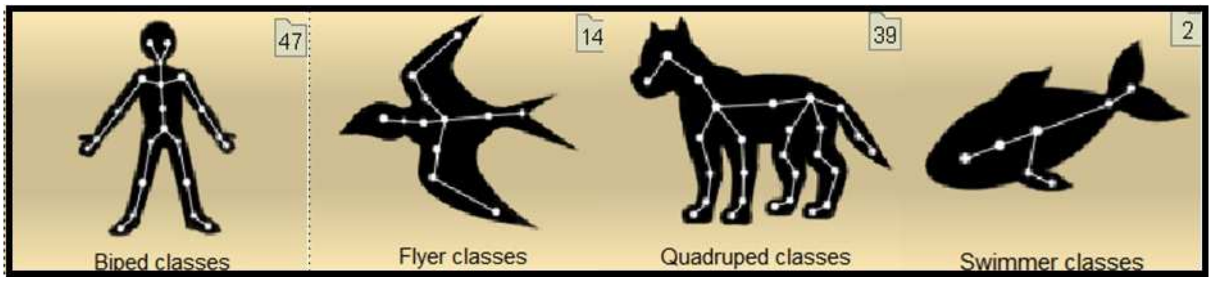
Contents

1 Scene Setting

1.1 Sub-parts and joints

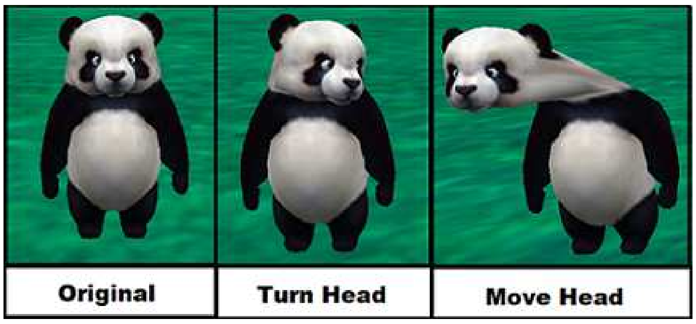
1.1.1 Joints and Movements

Some of the Alice classes have joints, and these joints can be turned to allow an object to move. The superclasses shown below define the joints an object will have. Some subclasses may have additional joints.



To program realistic animation, you need to look carefully at what actually happens in real life. For example, to lift your am, what you actually do is turn your shoulder joint. Spend a bit of time watching people and animals to see how they move. Also notice the difference between quadruped joints and biped joints. A cat's ankle, for example, is hallway up its back leg, and the knee is right at the top of the leg.

In Alice, you would always use a 'turn' method rather than a 'move' method to change the position of body parts, as this is what happens in real life. Alice does allow you to move a body part, but the results are rather strange, and often very funny. See the image below for an example.

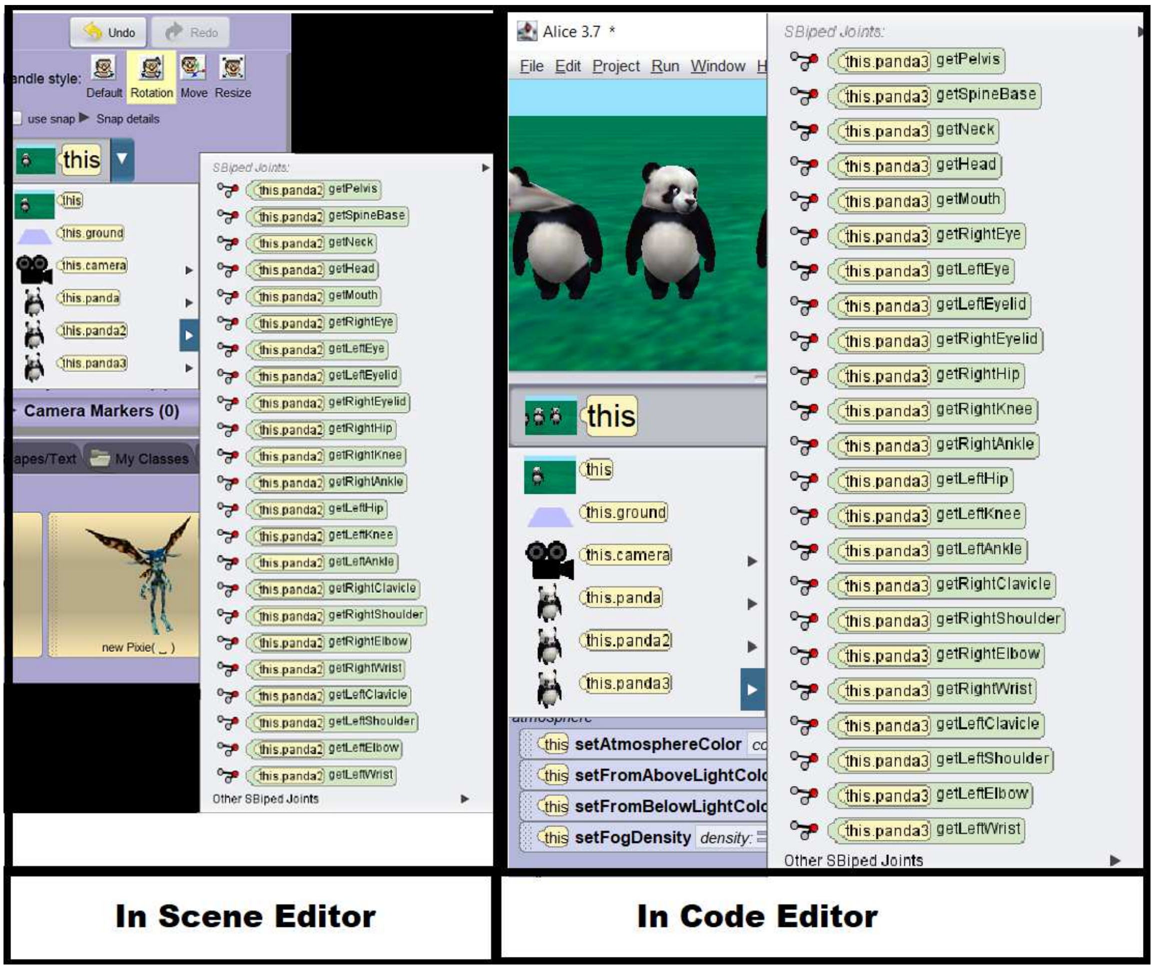


Sub-parts always turn according to their own orientation, and their orientation may be different to that of the object they belong to. For example, to raise the right leg forwards, you would in fact turn the right hip backwards. Remember that for a turn, a value of 1 is a full circle; to turn 90 degrees, use 0.25. Moving sub-parts is best learnt by experiment. You can always use the undo button if you get it wrong!

1.1.2 Using methods of sub-parts

Sub-parts of an object have their own methods, which can be used both in your code and in one shots. First you would select the sub part, and then you can use its methods.

To select a sub part, click the object selector. From the list that appears below it, click the arrow next to the object whose sub-part you want to access. A list of sub-parts will then appear, and you can click the part you want to work with.



The sub-part will then become the current object, and you can use its methods as you would for any other object.

1.2 Thinking in 3D

1.2.1 3D Considerations

The Alice world and all the objects in it are three dimensional. They have height, width and depth, and their position is defined by three axes known as x, y and z. You need to always be aware of where an object is in three dimensions.

1.2.2 Placing one object on another

Placing one object on, or inside, another, can be tricky in 3D. You may need to do this if you want to make one of your characters ride a horse or drive in a car. I suggest this method, using the example of a person sitting in a boat:

a) Create both objects apart from each other, but make sure they are at the same Z location (in other words, the same distance from the camera). You can use properties to achieve this. This lets you check if they are a good match for size. Adjust their sizes if you need to.
b) Use one-shots to move and orient the sitter to the boat.
c) Move the sitter up to the correct height using the move handle.
d) Position body parts of the sitting person to the right pose.
e) Set the person's vehicle to the boat.
f) In the code editor, write instructions to turn the boat right for a full turn on a count of 10, and then forward for a full turn for a count of 10. Set the opacity of the ground to 0, so you can still see the object when it turns underground. Run the program; this will allow you to see if the two objects look right from all angles.
g) Make any adjustments and test as needed. Finally, set the opacity of the ground back to 1.

2 More complex programs

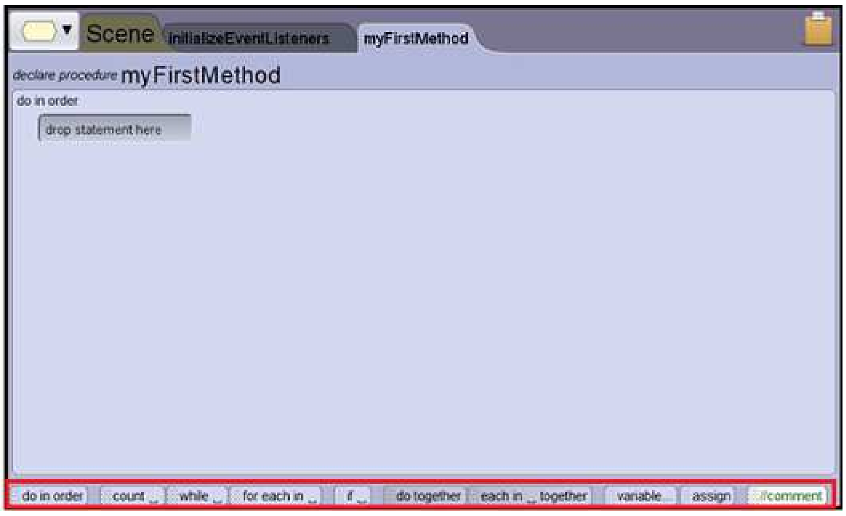
2.1 Programming Constructs

Programming constructs give you more control of the program.

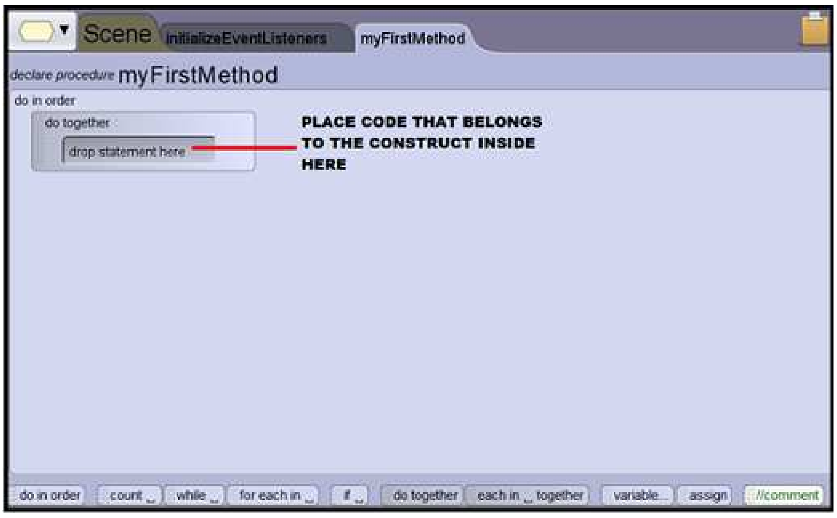
In programming, there are four different kinds of logic constructs:

Sequence:	Actions are carried out one at a time in order.
Iteration:	A set of actions is repeated
Conditional:	A set of actions is only carried out when a defined condition is met
Concurrency:	Two or more actions are carried out at the same time

The Alice code editor has several available constructs at the bottom of the screen, as seen in the image below highlighted in red. These can be used to implement the four constructs mentioned above, as well as other functionality.



To use them, drag them into the code editor. In the image below, the 'do together' construct has been dragged in, and other instructions can be placed inside it.

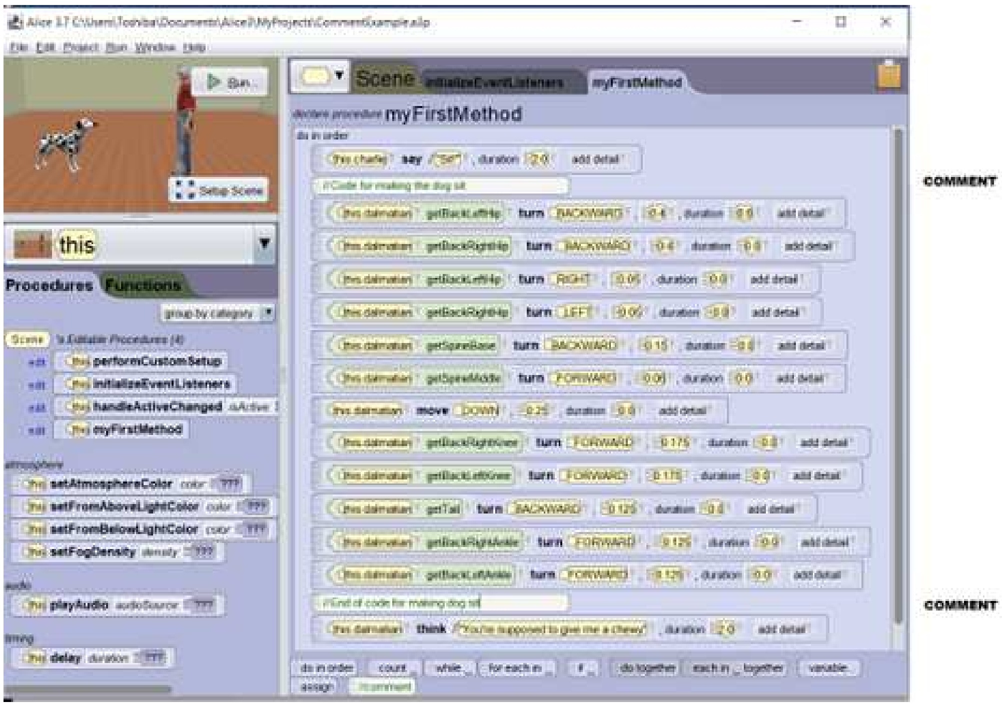


2.1.1 Comments

When you write large programs, you will often find that when you come back after a period of time to add to it or change it, you can't remember what a particular piece of code was for. Also, if you work together with someone else on a large program, they may not understand what you're doing with a section of code.

Comments make code a lot easier to understand. They don't affect the way the program runs.

Drag in a comment box and into it type any notes that you think may be helpful in understanding the program.


Alice 3.7 C:\Users\Toshiba\Documents\Alice3\MyProjects\CommentExample.a3p
File Edit Project Run Window Help
Run
Setup Scene
this
Procedures Functions
group by category
Scene 's Editable Procedures (4)
edit this performCustomSetup
edit this initializeEventListeners
edit this handleActiveChanged isActive
edit this myFirstMethod
atmosphere
this setAtmosphereColor color ???
this setFromAboveLightColor color ???
this setFromBelowLightColor color ???
this setFogDensity density ???
audio
this playAudio audioSource ???
timing
this delay duration ???
COMMENT
COMMENT

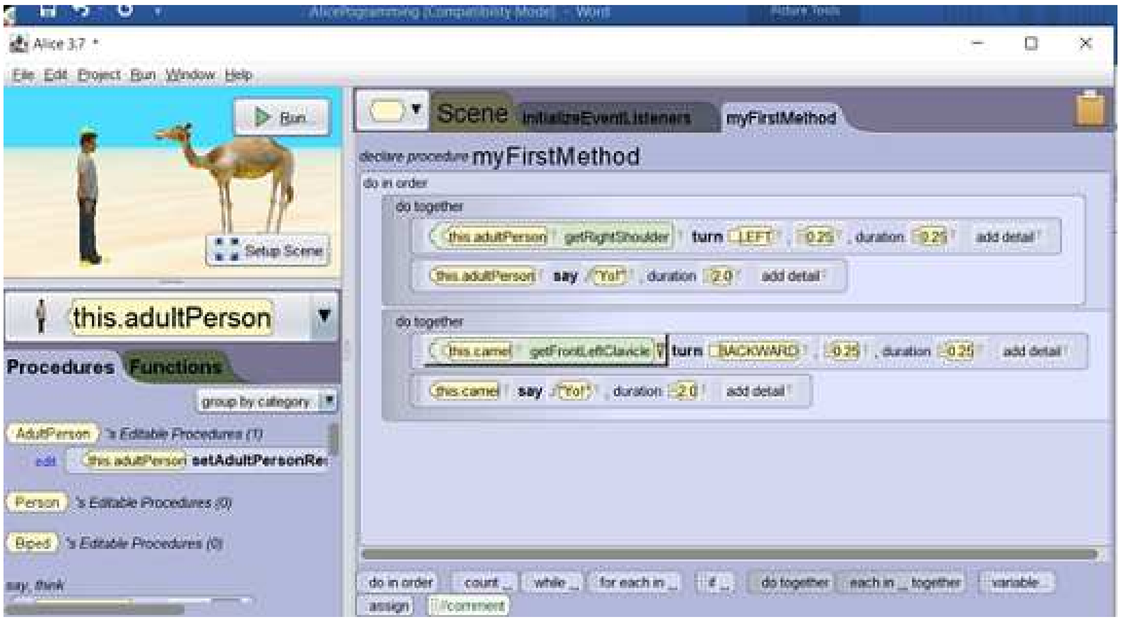
```
declare procedure myFirstMethod
do in order
    this.charlie say "Sit!" , duration 2.0   add detail
    //Code for making the dog sit
    this.dalmatian getBackLeftHip turn BACKWARD , 0.4 , duration 0.0   add detail
    this.dalmatian getBackRightHip turn BACKWARD , 0.4 , duration 0.0   add detail
    this.dalmatian getBackLeftHip turn RIGHT , 0.05 , duration 0.0   add detail
    this.dalmatian getBackRightHip turn LEFT , 0.05 , duration 0.0   add detail
    this.dalmatian getSpineBase turn BACKWARD , 0.15 , duration 0.0   add detail
    this.dalmatian getSpineMiddle turn FORWARD , 0.08 , duration 0.0   add detail
    this.dalmatian move DOWN , 0.25 , duration 0.0   add detail
    this.dalmatian getBackRightKnee turn FORWARD , 0.175 , duration 0.0   add detail
    this.dalmatian getBackLeftKnee turn FORWARD , 0.175 , duration 0.0   add detail
    this.dalmatian getTail turn BACKWARD , 0.125 , duration 0.0   add detail
    this.dalmatian getBackRightAnkle turn FORWARD , 0.125 , duration 0.0   add detail
    this.dalmatian getBackLeftAnkle turn FORWARD , 0.125 , duration 0.0   add detail
    //End of code for making dog sit
    this.dalmatian think "You're supposed to give me a chewy" , duration 2.0   add detail
```

2.1.2 Do in order and do together

So far, we have only looked at actions that are done in order, one after the other.

You may have noticed that the action in some of the earlier practicals was not smooth. There were delays between each step of the action, for example, when a sound was played, or when the 'say' and 'think' methods were used.

This can be improved by using a 'do together' construct when needed. The 'do together' is used to achieve the concurrency programming construct. For example, the 'say' instruction could be placed in a 'do together' block with the action, so that the character speaks at the same time as they move. Drag in a 'do together' block, and place the actions that should take place at the same time inside it. See the example in the image below.



```
declare procedure myFirstMethod
do in order
    do together
        (this.adultPerson getRightShoulder) turn LEFT, 0.25, duration 0.25  add detail
        this.adultPerson say "Yo!", duration 2.0  add detail
    do together
        (this.camel getFrontLeftClavicle) turn BACKWARD, 0.25, duration 0.25  add detail
        this.camel say "Yo!", duration 2.0  add detail
```

Most movements of body parts to achieve a task are best done together, for example when doing a star jump, the arms and legs move at the same time as the body moves up. This can easily be achieved by using a 'do together.'

You would also use it if you wanted more than one character to do the same moves at the same time, for example, a row of characters doing a line dance.

It's worth taking the time to plan which actions should be done together, and which in sequence.

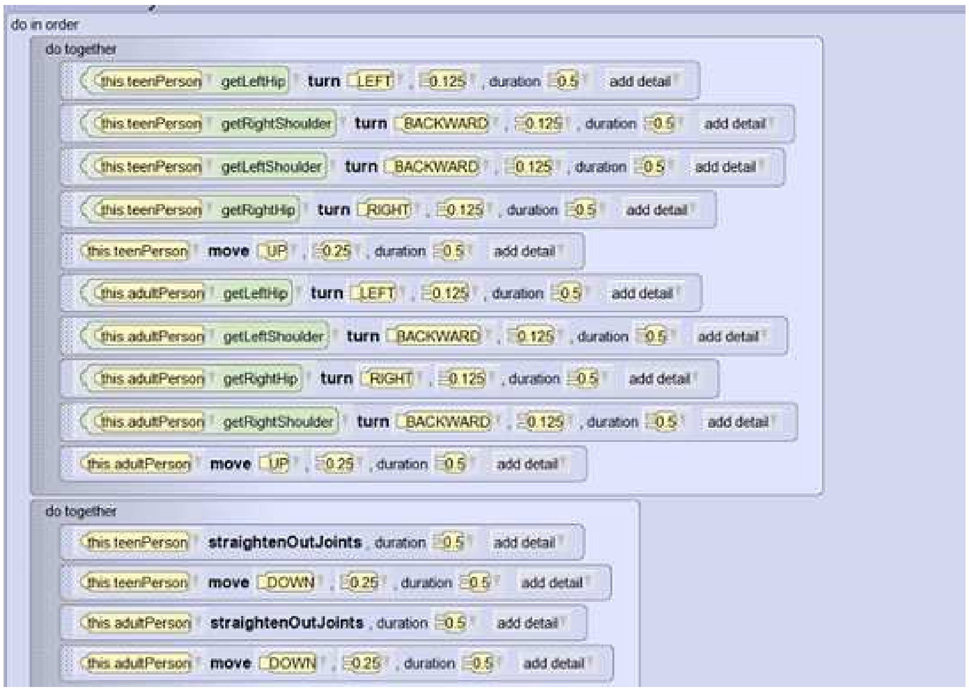
For example, if you wanted to play a sound track while characters carried out a dance step, your logic might look like this:

```
Do Together:
    Play Audio
    Do in Order
        Do Together
            All characters do Step 1
        Do Together
            All characters do Step 2
        Do Together
            All characters do Step 3
```

2.1.3 Count loops

Count loops are one form of the iteration (repetition) construct. You would use them if you wanted a section of code repeated a fixed number of times.

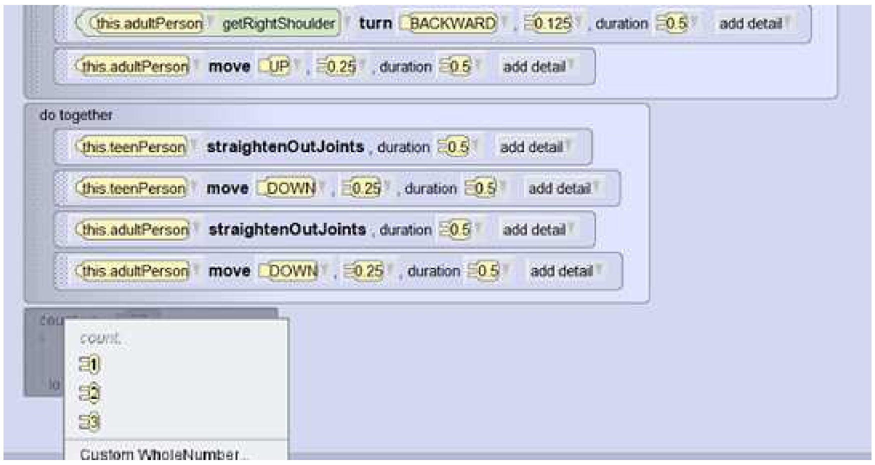
For example, if we've written code to make two characters do star jumps, we might decide to make them do the jumps three times. The original code might look like this:



```
do in order
  do together
    (this.teenPerson getLeftHip) turn LEFT , 0.125 , duration 0.5 add detail
    (this.teenPerson getRightShoulder) turn BACKWARD , 0.125 , duration 0.5 add detail
    (this.teenPerson getLeftShoulder) turn BACKWARD , 0.125 , duration 0.5 add detail
    (this.teenPerson getRightHip) turn RIGHT , 0.125 , duration 0.5 add detail
    this.teenPerson move UP , 0.25 , duration 0.5 add detail
    (this.adultPerson getLeftHip) turn LEFT , 0.125 , duration 0.5 add detail
    (this.adultPerson getLeftShoulder) turn BACKWARD , 0.125 , duration 0.5 add detail
    (this.adultPerson getRightHip) turn RIGHT , 0.125 , duration 0.5 add detail
    (this.adultPerson getRightShoulder) turn BACKWARD , 0.125 , duration 0.5 add detail
    this.adultPerson move UP , 0.25 , duration 0.5 add detail
  do together
    this.teenPerson straightenOutJoints , duration 0.5 add detail
    this.teenPerson move DOWN , 0.25 , duration 0.5 add detail
    this.adultPerson straightenOutJoints , duration 0.5 add detail
    this.adultPerson move DOWN , 0.25 , duration 0.5 add detail
```

To make them jump three times, we'd need to put this code inside a 'count' block.

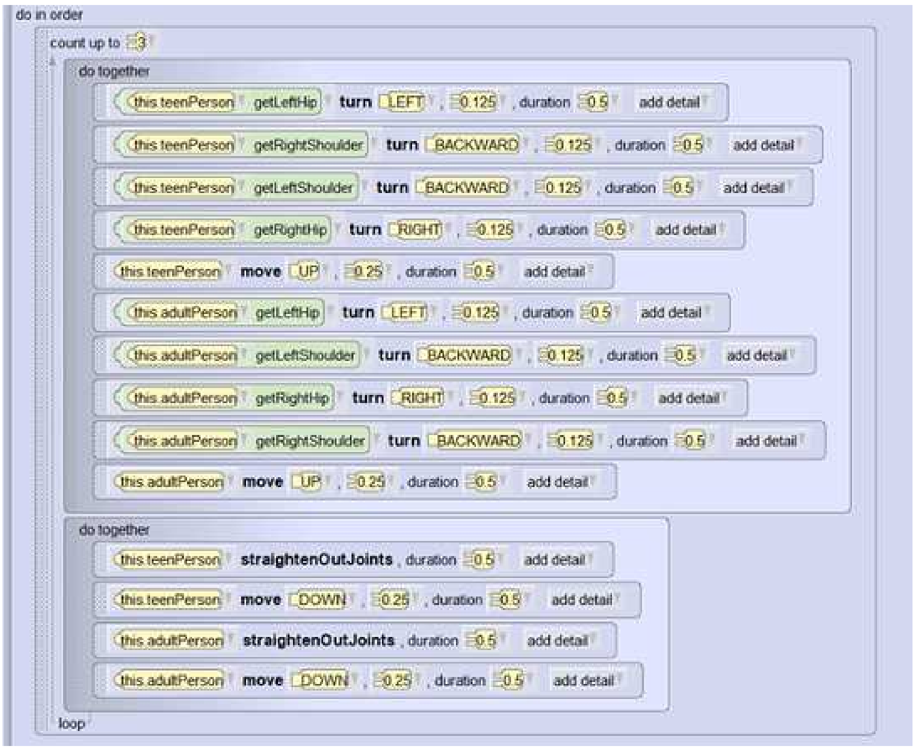
First we drag the count block into the code. We'll need to give a parameter telling it how many jumps we want.



```
    (this.adultPerson getRightShoulder) turn BACKWARD , 0.125 , duration 0.5 add detail
    this.adultPerson move UP , 0.25 , duration 0.5 add detail
  do together
    this.teenPerson straightenOutJoints , duration 0.5 add detail
    this.teenPerson move DOWN , 0.25 , duration 0.5 add detail
    this.adultPerson straightenOutJoints , duration 0.5 add detail
    this.adultPerson move DOWN , 0.25 , duration 0.5 add detail
```

We choose 3.

Next we drag the existing code blocks inside the count block, so that the final code looks like this:



```
do in order
  count up to 3
    do together
      this.teenPerson.getLeftHip turn LEFT , 0.125 , duration 0.5 add detail
      this.teenPerson.getRightShoulder turn BACKWARD , 0.125 , duration 0.5 add detail
      this.teenPerson.getLeftShoulder turn BACKWARD , 0.125 , duration 0.5 add detail
      this.teenPerson.getRightHip turn RIGHT , 0.125 , duration 0.5 add detail
      this.teenPerson move UP , 0.25 , duration 0.5 add detail
      this.adultPerson.getLeftHip turn LEFT , 0.125 , duration 0.5 add detail
      this.adultPerson.getLeftShoulder turn BACKWARD , 0.125 , duration 0.5 add detail
      this.adultPerson.getRightHip turn RIGHT , 0.125 , duration 0.5 add detail
      this.adultPerson.getRightShoulder turn BACKWARD , 0.125 , duration 0.5 add detail
      this.adultPerson move UP , 0.25 , duration 0.5 add detail
    do together
      this.teenPerson straightenOutJoints , duration 0.5 add detail
      this.teenPerson move DOWN , 0.25 , duration 0.5 add detail
      this.adultPerson straightenOutJoints , duration 0.5 add detail
      this.adultPerson move DOWN , 0.25 , duration 0.5 add detail
  loop
```

The jumps will then be repeated three times when we run the program.

2.1.4 Variables

Variables are places where we can store information in the program. You can think of them being like little boxes, each one holding some information.

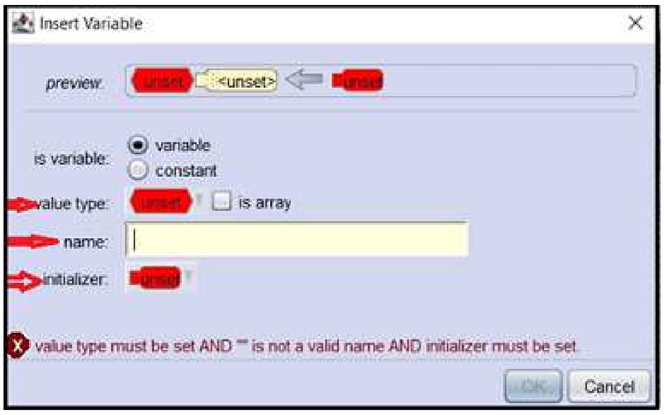
One of the common uses for variables is to store a user's answer to a question, then use the answer later in the program.

For example, in the star jumps program, we could ask the user how many times the characters should jump, then use his answer as the parameter to the 'count' loop.

2.1.4.1 Creating a variable

The first step is to define the variable by dragging 'variable' from the bottom of the code screen into the code. Variables must be placed in the code above the code where they will be used. Some people like to put all the variables at the top of the code.

When you drag 'variable' into the code, a box will pop up asking for information about the variable.



Insert Variable

preview: unset <unset> ⟸ unset

is variable: [✓ variable] constant

value type: unset ☐ is array

name:

initializer: unset

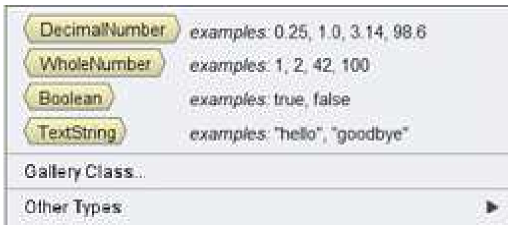
value type must be set AND "" is not a valid name AND initializer must be set

OK Cancel

You'll need to fill in the three sections marked with arrows in the image.

Value Type:

All variables must have a value type. This defines what type of information they will store. Clicking the arrow next to the value type pops up a box where you can choose from the available types:



Data types available are:

- Decimal number – a number which allows decimal points e.g. 3.25
- Whole number – a number which does not allow decimal points, e.g. 3
- Boolean – can hold a value to indicate true or false; used with conditions
- Text String – this can hold any value that can be typed in from the keyboard, e.g. the users's name.
- Gallery class – stores a pointer to an object of the selected class, e.g. Biped
- Other types – stores miscellaneous Alice information, e.g. a move direction

Name:

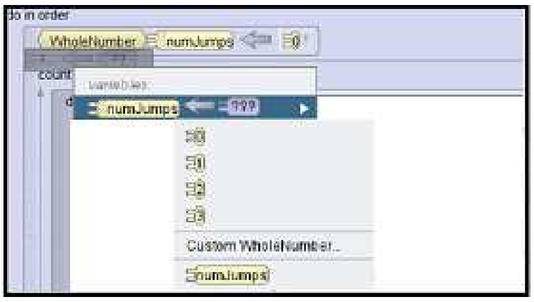
All variables must have a name. They follow the same rules as object names.

Initial value:

This is the value that the variable will have when it's first created. If the variable is to be given a value later, for example by asking the user to enter it, you can set the initial value to anything; perhaps 0 for a number.

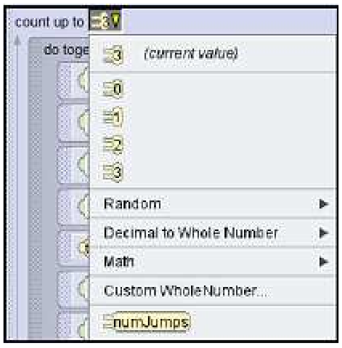
2.1.4.2 Assigning a value to a variable

To assign a value to a variable, drag 'assign' from the bottom of the code screen. As per the image below, this will require two parameters: the name of the variable that the value will be assigned to, and the value to be assigned. It's possible to assign an actual value, or to set it to have the same value as another variable, if one is available.



2.1.4.3 Using a variable in a Count loop

Variables can be used as parameters to most Alice procedures. If we wanted to change the star jumps program to count up to whatever number is stored in the **numJumps** variable, we can click the arrow next to 'count up to 3'. The new variable will show up as one of the possible choices, as seen in the image below.

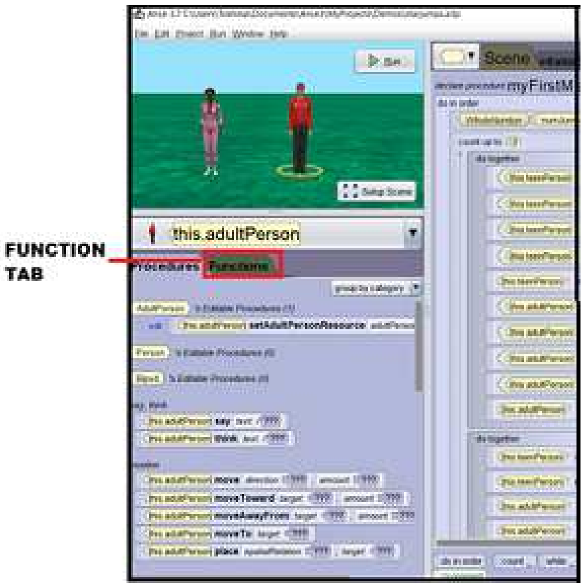


Click **numJumps** to make the change.

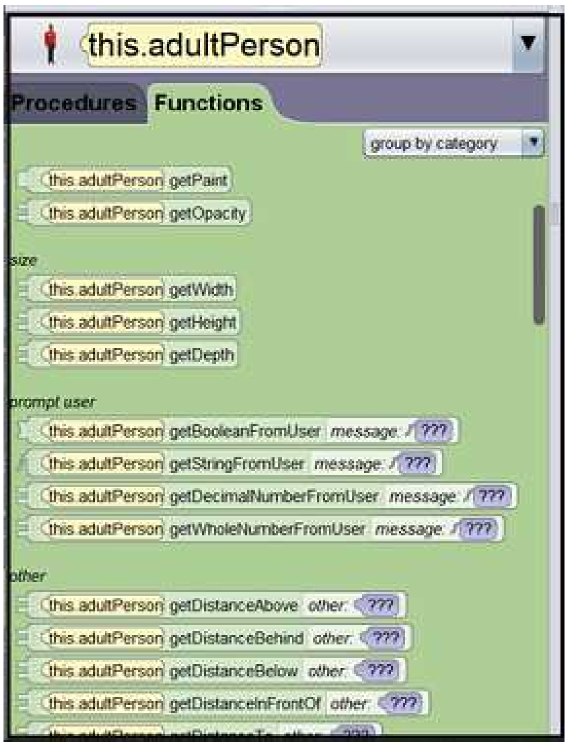
2.1.4.4 Assigning the result of a function to a variable

There are two types of methods: procedures and functions. Procedures perform a task, whereas functions retrieve a value. We can therefore use functions to assign a value to a variable.

Just as objects have procedures, they also have functions. To access an object's functions instead of its procedures, click the function tab as shown in the image below.



This will then show the list of available functions for the object.



this.adultPerson

Procedures Functions

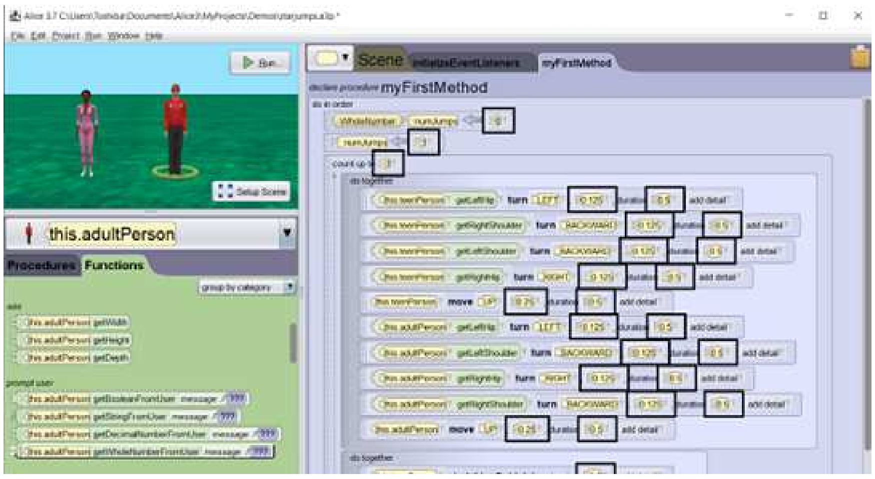
group by category

```
this.adultPerson getPaint
this.adultPerson getOpacity
size
this.adultPerson getWidth
this.adultPerson getHeight
this.adultPerson getDepth
prompt user
this.adultPerson getBooleanFromUser message: ???
this.adultPerson getStringFromUser message: ???
this.adultPerson getDecimalNumberFromUser message: ???
this.adultPerson getWholeNumberFromUser message: ???
other
this.adultPerson getDistanceAbove other: ???
this.adultPerson getDistanceBehind other: ???
this.adultPerson getDistanceBelow other: ???
this.adultPerson getDistanceInFrontOf other: ???
```

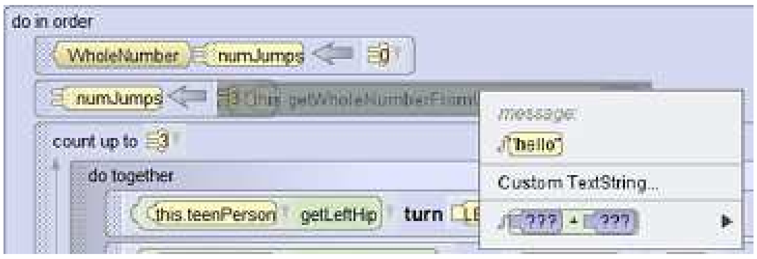
Any of these can be used in an assignment to set the value of a variable, provided the function and the variable have the same data type. For example, the getStringFromUser function could be used to place a value in a variable that has been defined as a text string, but not one that has been defined as a number.

If we wanted to ask the user how many star jumps the characters should do, and place the answer in the numJumps variable, we would need the getWholeNumberFromUser function, since numJumps is a whole number.

If we hover the mouse over this function, Alice highlights all the places in the program where we would be allowed to place the result of this function, as per the image below.

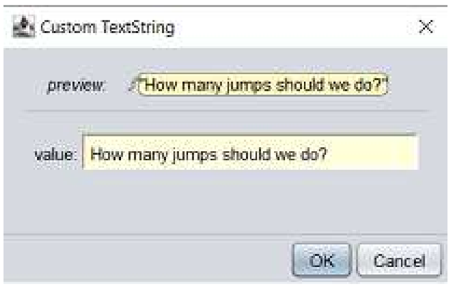


Alice shows a black rectangle around each of the places where we could use the function. Dragging the function to the correct place – in this case, the assignment to **numJumps,** pops up the options shown in the image below.

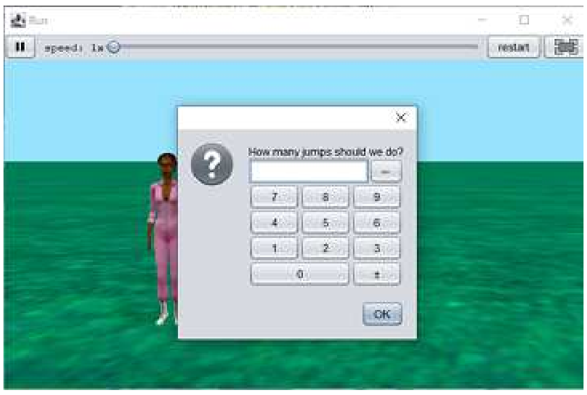


The **getWholeNumberFromUser** function requires one parameter: a text string that should contain the message to the user asking him a question which he must answer.

We can choose Custom TextString, and type in a meaningful message, for example, 'How many jumps should we do?'


Custom TextString
preview: "How many jumps should we do?"
value: How many jumps should we do?
OK
Cancel

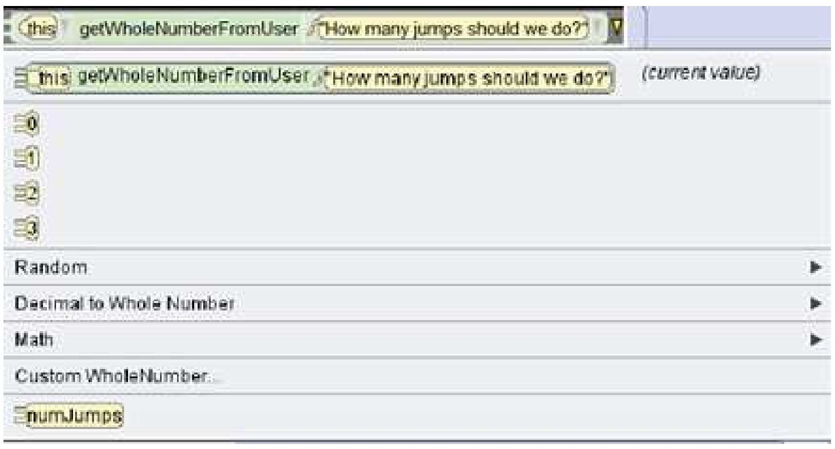
When the program is run, it will pop up a box asking the user for input. Whatever he/she types will be stored in **numJumps**, and will control the number of jumps.



2.1.4.5 Using mathematical expressions in assignments

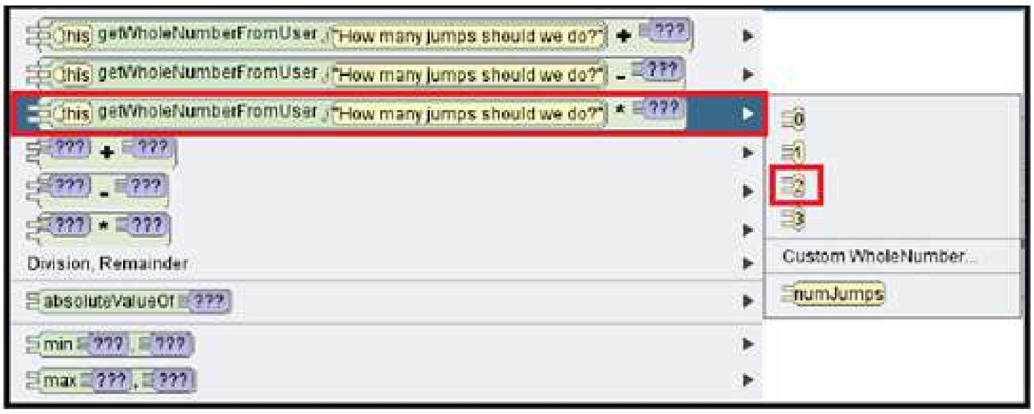
You can include mathematical calculations in assignments. Suppose we wanted the characters to do twice as many jumps as the user requested.

If we click the arrow next to the variable assignment, we get the following pop-up:


this getWholeNumberFromUser "How many jumps should we do?"
this getWholeNumberFromUser "How many jumps should we do?" (current value)
0
1
2
3
Random
Decimal to Whole Number
Math
Custom WholeNumber...
numJumps

To carry out a calculation, click Math.

The Options pop up as seen in the image below. To set the variable to the user's answer multiplied by 2, choose the options highlighted in red.


this getWholeNumberFromUser "How many jumps should we do?" + ???
this getWholeNumberFromUser "How many jumps should we do?" - ???
this getWholeNumberFromUser "How many jumps should we do?" * ???
??? + ???
??? - ???
??? * ???
Division, Remainder
absoluteValueOf ???
min ???, ???
max ???, ???
0
1
2
3
Custom WholeNumber...
numJumps

2.1.5 Conditionals

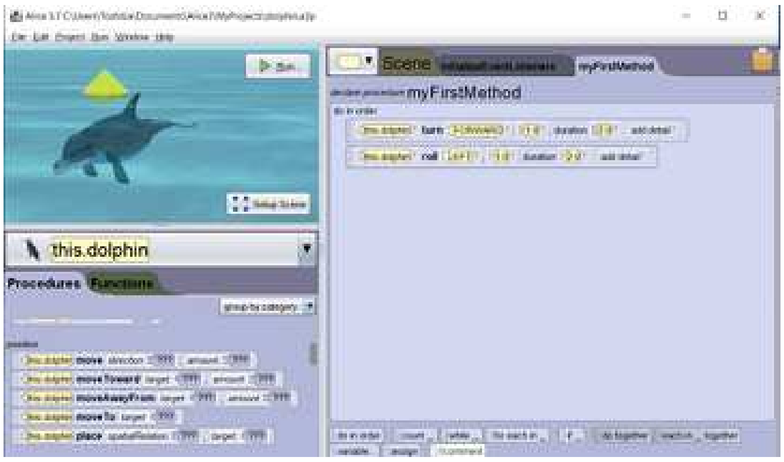
Conditionals allow us to control whether or not an instruction should be carried out depending on a condition. This is something we do often in real life, for example,

- If it's sunny I will sit in the garden
- If I'm hungry I will have a sandwich
- If you're late I won't wait for you.

Sometimes, also in real life, we may have an alternative action if the condition is not met, for example, 'If it's sunny I will sit in the garden, otherwise, I will watch TV.'

In Alice, we can use an 'if…else' construct to make decisions in our program.

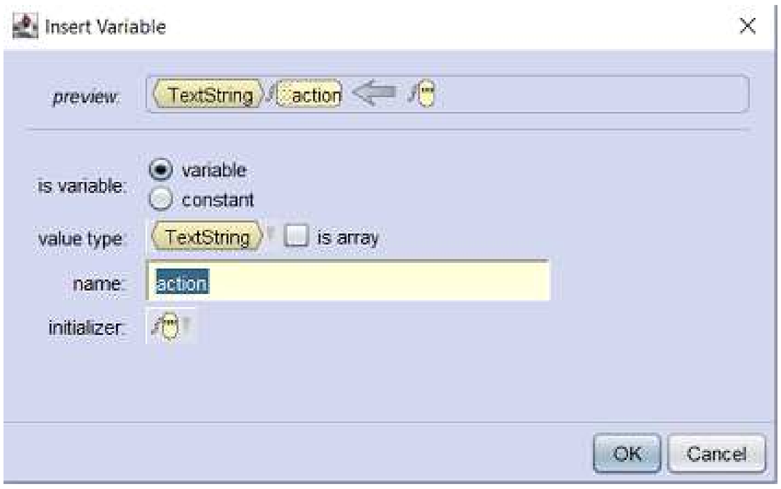
In the example below, we have a dolphin that does a forward flip, then a side roll.



```
declare procedure myFirstMethod
do in order
    this.dolphin turn FORWARD 1.0 duration 1.0 add detail
    this.dolphin roll LEFT 1.0 duration 2.0 add detail
```

We want to change it so the user can choose which action the dolphin should do. If the user enters 'Flip', the dolphin will do the flip; if the user enters 'Roll', he will do the roll. If the user enters anything else, he will say, 'Sorry, I don't know how to do that.'

Firstly we need a variable to store the user's answer. We'll set the variable's initial value to an empty text string.



Insert Variable

preview: TextString action ⟸ ""

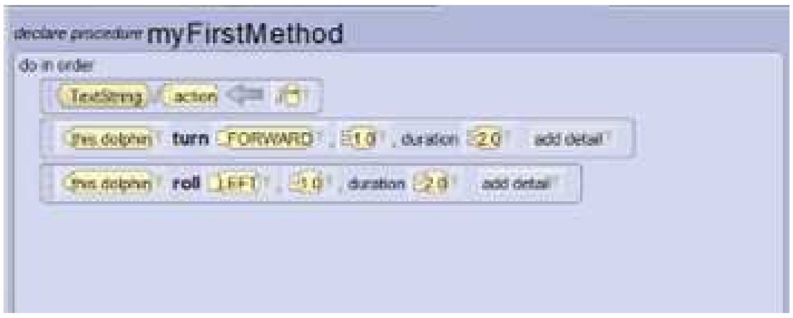
is variable: (●) variable ( ) constant

value type: TextString [ ] is array

name: action

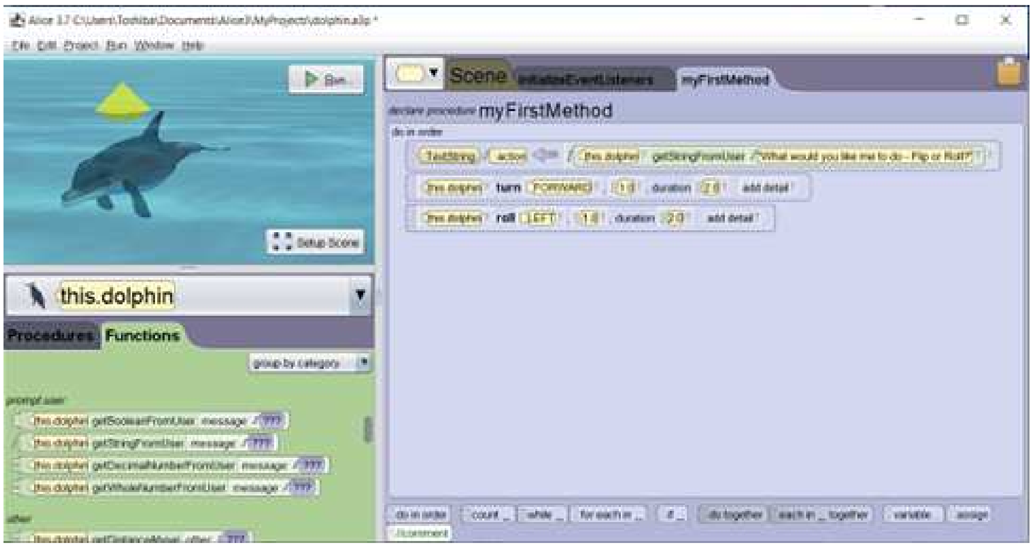
initializer: ""

The code will look like this:



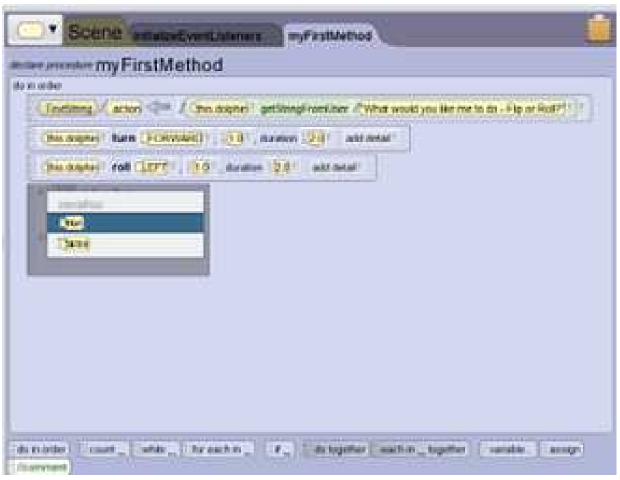
```
declare procedure myFirstMethod
do in order
    TextString action ⟸ ""
    this.dolphin turn FORWARD 1.0 , duration 2.0 add detail
    this.dolphin roll LEFT 1.0 , duration 2.0 add detail
```

Next we will request a text string from the user, and place the answer in the variable instead of the empty text string. We will use the 'getStringFromUser' function and drag it into the variable definition. The code now looks like this:



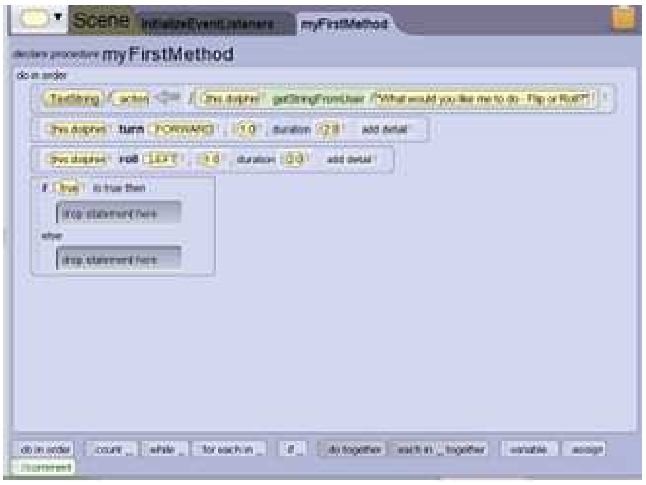
```
declare procedure myFirstMethod
do in order
    TextString action ⇦ this.dolphin getStringFromUser "What would you like me to do - Flip or Roll?"
    this.dolphin turn FORWARD , 1.0 , duration 2.0 add detail
    this.dolphin roll LEFT , 1.0 , duration 2.0 add detail
```

Next we drag in the ‘if’ construct from the bottom of the screen. This gives us the choice shown below between ‘true’ or ‘false’.



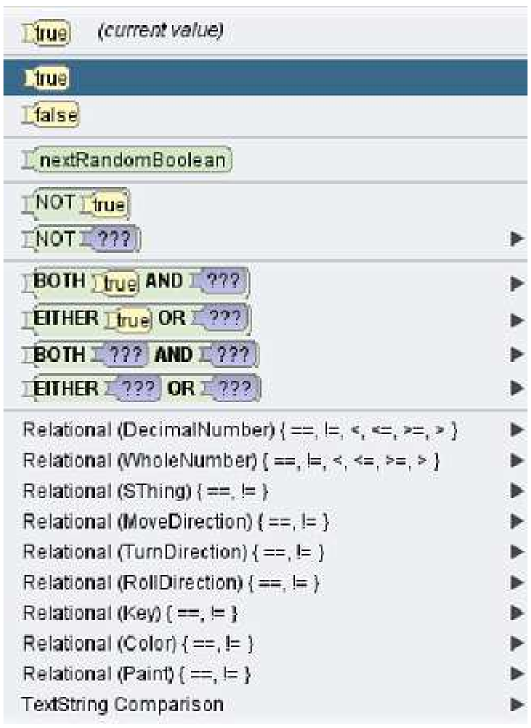
```
declare procedure myFirstMethod
do in order
    TextString action ⇦ this.dolphin getStringFromUser "What would you like me to do - Flip or Roll?"
    this.dolphin turn FORWARD , 1.0 , duration 2.0 add detail
    this.dolphin roll LEFT , 1.0 , duration 2.0 add detail
```

We choose ‘true’, and the code looks like this:

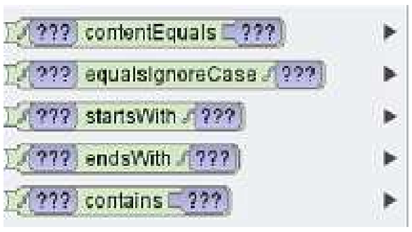


```
declare procedure myFirstMethod
do in order
    TextString action ⇦ this.dolphin getStringFromUser "What would you like me to do - Flip or Roll?"
    this.dolphin turn FORWARD , 1.0 , duration 2.0 add detail
    this.dolphin roll LEFT , 1.0 , duration 2.0 add detail
    if true is true then
        drop statement here
    else
        drop statement here
```

We now change the condition to the condition we actually want, which is to check the content of the variable named **action.** Click the arrow next to the word ‘true’. This gives us the following:

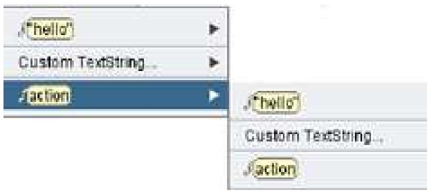

true (current value)
true
false
nextRandomBoolean
NOT true
NOT ???
BOTH true AND ???
EITHER true OR ???
BOTH ??? AND ???
EITHER ??? OR ???
Relational (DecimalNumber) { ==, !=, <, <=, >=, > }
Relational (WholeNumber) { ==, !=, <, <=, >=, > }
Relational (SThing) { ==, != }
Relational (MoveDirection) { ==, != }
Relational (TurnDirection) { ==, != }
Relational (RollDirection) { ==, != }
Relational (Key) { ==, != }
Relational (Color) { ==, != }
Relational (Paint) { ==, != }
TextString Comparison

These are all the different types of conditions we can have, In this case, we want to look at the value of the TextString type variable **action**, so it will be a TextString Comparison. Moving the cursor down to this option gives us:



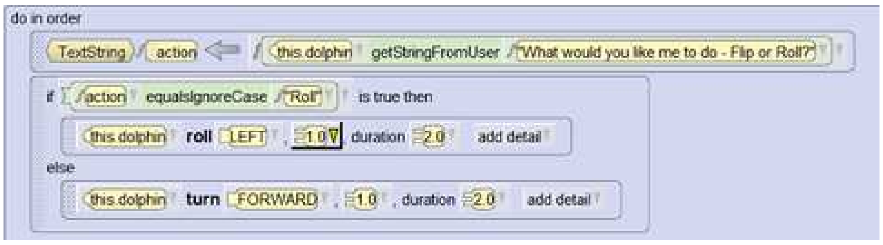
These are the type of comparisons allowed in the condition. For this program, we could choose ‘equalsIgnoreCase’, which means that if our user typed ‘Roll’, it would count as being the same as ‘roll’ without the capital letter.

We then choose the two things we want to compare:



In this case, we want to compare the contents of the variable **action** with a custom text string containing ‘Roll’.

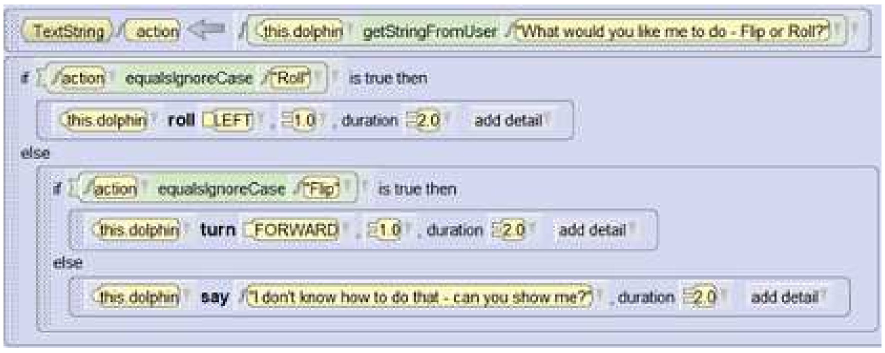
Once we have defined the condition, we can drag the existing code into the correct place within the ‘if’ box, so that the code now looks like this:



```
do in order
  TextString action <- this.dolphin getStringFromUser "What would you like me to do - Flip or Roll?"
  if action equalsIgnoreCase "Roll" is true then
    this.dolphin roll LEFT , 1.0 duration 2.0 add detail
  else
    this.dolphin turn FORWARD , 1.0 , duration 2.0 add detail
```

The program now works. However, we can still improve on this, because we haven't allowed for all possibilities. If the user answered something other than 'Flip' or 'Roll' (maybe 'Jump'), the dolphin will still do the flip, because the user didn't enter roll. So we could add an additional 'if' box to check if he actually did type 'Flip' if the first condition is not true.

The code could look like this:



```
TextString action <- this.dolphin getStringFromUser "What would you like me to do - Flip or Roll?"
if action equalsIgnoreCase "Roll" is true then
  this.dolphin roll LEFT , 1.0 , duration 2.0 add detail
else
  if action equalsIgnoreCase "Flip" is true then
    this.dolphin turn FORWARD , 1.0 , duration 2.0 add detail
  else
    this.dolphin say "I don't know how to do that - can you show me?" , duration 2.0 add detail
```

As a programmer, you should always try to think of all the possible things that could happen, and cater for them.

2.1.6 While loops

The 'while' loop is another form of iteration, or repetition, and it's probably the most widely used. It's similar to the 'count' loop, in that instructions are repeated, but instead of repeating a fixed number of times, they repeat while a condition remains true.

Some examples:

- A treasure hunt game may continue while the treasure has not been found
- A game with a spaceship fighting aliens may continue until the ship is destroyed.
- A game may be repeated while the user chooses to keep playing.

Conditions for a 'while' loop take the same form as conditions for an 'if'. The code inside the while box will keep being repeated until the condition becomes false.

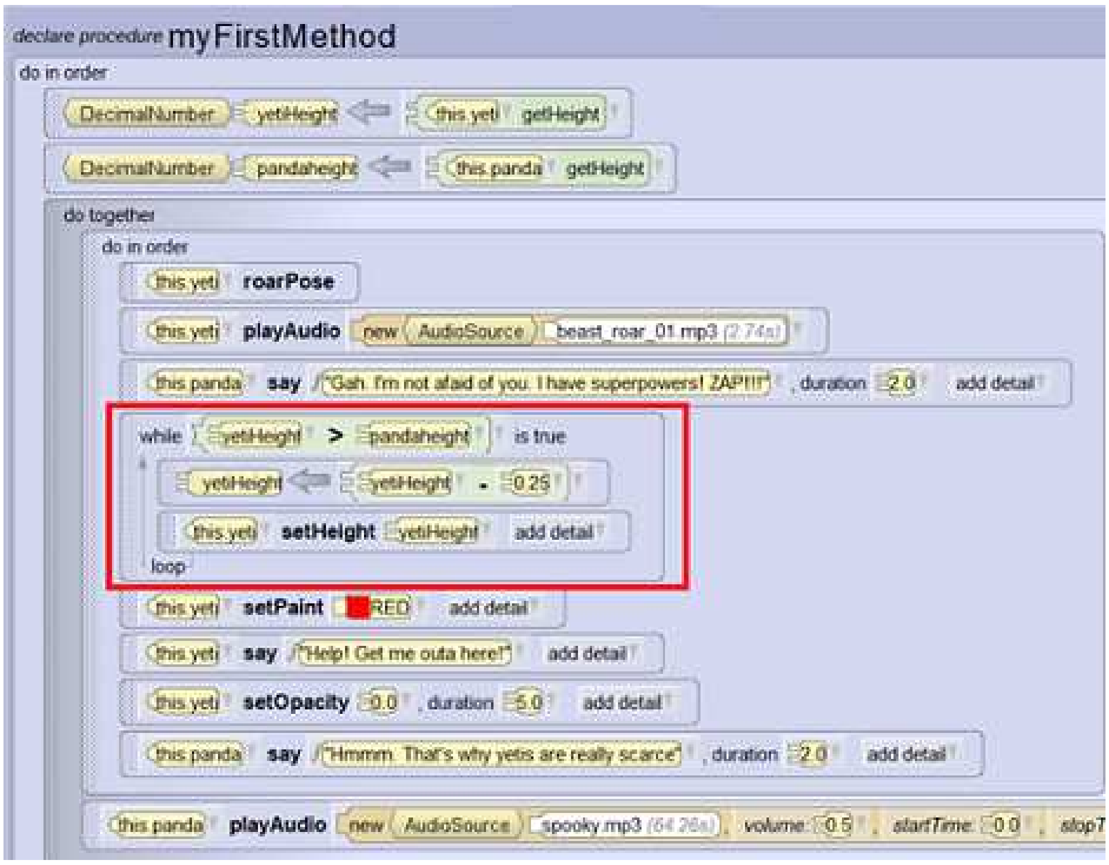
Below is an example of a program using a 'while' loop.

There are two characters: a yeti and a panda, as shown in the image.

The program's story is:

- The yeti threatens the panda and roars
- The panda says he's not afraid because he has superpowers, and zaps the yeti
- The yeti slowly shrinks until he is the same size as the panda.
- He turns red, shouts for help and then slowly fades away
- The panda says that's why yetis are scarce

The program uses a while loop to shrink the yeti; he keeps shrinking until he is the same height as the panda. It uses the function **getHeight** to get the heights of the panda and the yeti, and compares them in the condition that controls the while loop. It uses the procedure setHeight to make the yeti decrease in size within the loop.



```
declare procedure myFirstMethod
do in order
    DecimalNumber yetiHeight ⇐ this yeti getHeight
    DecimalNumber pandaheight ⇐ this panda getHeight
    do together
        do in order
            this yeti roarPose
            this yeti playAudio new AudioSource beast_roar_01.mp3 (2.74s)
            this panda say "Gah. I'm not afaid of you. I have superpowers! ZAP!!!" , duration 2.0   add detail
            while yetiHeight > pandaheight is true
                yetiHeight ⇐ yetiHeight - 0.25
                this yeti setHeight yetiHeight   add detail
            loop
            this yeti setPaint RED   add detail
            this yeti say "Help! Get me outa here!"   add detail
            this yeti setOpacity 0.0 , duration 5.0   add detail
            this panda say "Hmmm. That's why yetis are really scarce" , duration 2.0   add detail
        this panda playAudio new AudioSource spooky.mp3 (64.26s) , volume: 0.5 , startTime: 0.0 , stopT
```

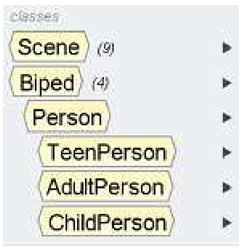
2.2 Writing procedures

2.2.1 Writing your own procedures

You've seen that Alice objects have procedures that can be called to make something happen to that object, for example moving and turning the objects. These procedures have been defined in the class.

You can also add your own procedures to a class. For example, you could write a procedure to make the object do star jumps. This procedure would include all of the instructions needed to achieve the jump.

It's important to add your procedure to the correct class in the hierarchy, depending on which objects you would like to be able to do star jumps. For example, if you have some people of different ages in your project, the class hierarchy might look like this:

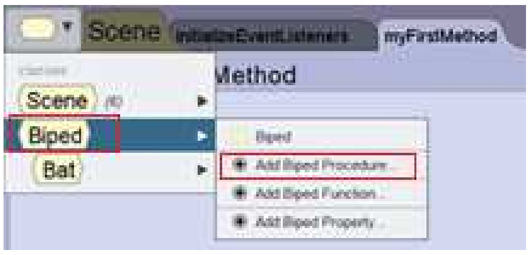


If you wanted all bipeds to be able to have a star jump procedure, you would add your new procedure to the **Biped** superclass. All your existing bipeds and any new ones you created would then have this procedure, which can be called either in one shots or in code.

If you added your procedure to the ChildPerson class, only objects created using the ChildPerson class would have the procedure.

To add a procedure to the Biped superclass:

- In the code editor, click on the navigator at the top
- Click Biped, and click Add Biped Procedure



- A code editor tab will open for it, and you can code the procedure.
- Note that when coding procedures in a class, 'this' always refers to the object of that class on which the actions will be carried out.

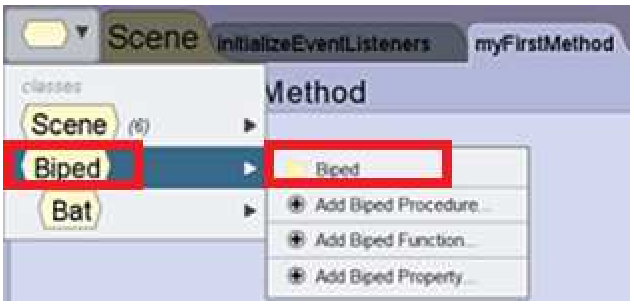
It's worth mentioning that the Biped superclass will only appear in the hierarchy if you already have a Biped in your project.

2.2.2 Exporting and importing procedures

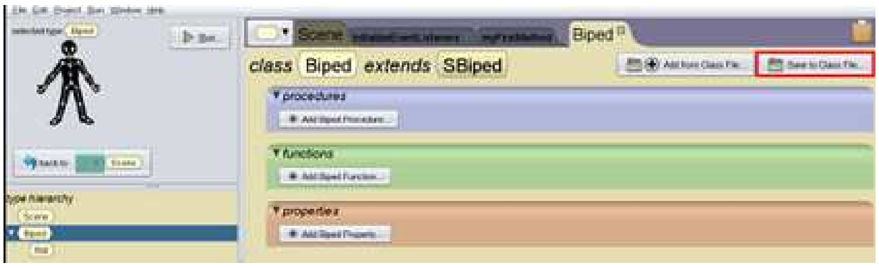
If you have written a useful procedure, you will very often want to reuse it in other projects. This is done by exporting the class containing the procedure. It can then be imported into any other project.

To export a procedure:

- Click on the Navigator in the coding editor, and click on the name of the class you are exporting, e.g. Biped, then from the resulting menu. Click Biped again.



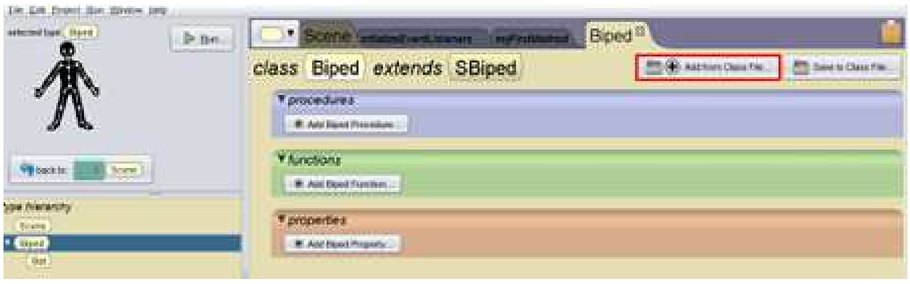
- The following screen will open



- Click 'Save to class file' as shown highlighted in red.
- Choose where and under what name you want to save it; the default is the name of the class, saved under your Alice MyClasses folder.

To import methods from this saved class into another project:

- Open the new project and create an object of the same class as the one you exported (e.g. Biped).
- Click on the navigator, and select the class
- The same screen as you used for exporting will appear. This time, click Add from Class File as highlighted.



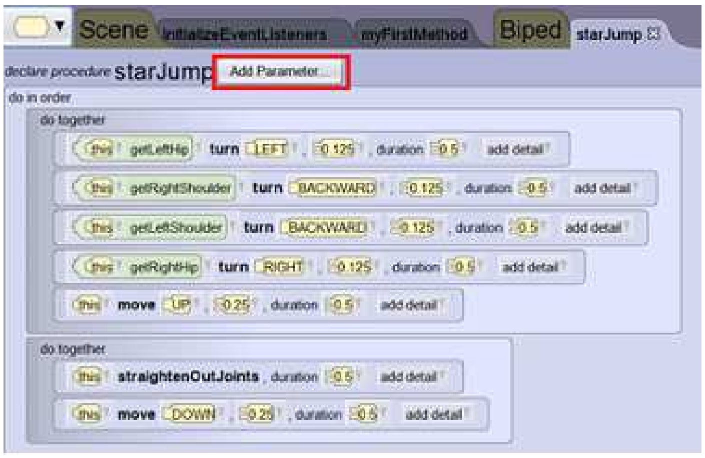
- Browse to where you saved the class, and select it
- A further screen will appear, allowing you to select which procedures you want to import. Make sure the right ones are ticked, click Next and then Finish.
- These procedures will then be available to be used in your new project

2.3 Using parameters with procedures

You will have seen that the procedures predefined in Alice often need parameters. For example, the MOVE procedure needs two parameters: the direction of the move and the distance.

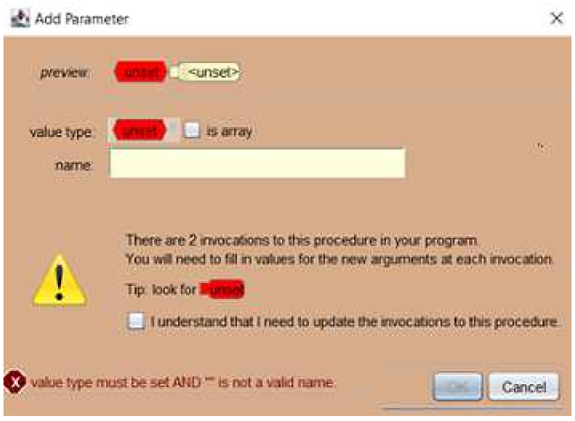
In the same way, you can include parameters in your own procedures. If you had written a procedure belonging to the Biped class to make the biped do a star jump, you may want to add a parameter to control how many jumps to perform.

To add a parameter to your procedure, click the 'Add Parameter' button when you have your procedure open in the code editor, as highlighted in red in the image below.



```
declare procedure starJump  Add Parameter...
do in order
    do together
        this.getLeftHip turn LEFT , 0.125 , duration 0.5  add detail
        this.getRightShoulder turn BACKWARD , 0.125 , duration 0.5  add detail
        this.getLeftShoulder turn BACKWARD , 0.125 , duration 0.5  add detail
        this.getRightHip turn RIGHT , 0.125 , duration 0.5  add detail
        this move UP , 0.25 , duration 0.5  add detail
    do together
        this straightenOutJoints , duration 0.5  add detail
        this move DOWN , 0.25 , duration 0.5  add detail
```

This will pop up a window to allow you to define your parameter:



There are 2 invocations to this procedure in your program.
You will need to fill in values for the new arguments at each invocation.

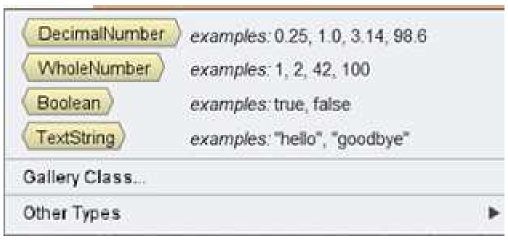
Tip: look for unset

I understand that I need to update the invocations to this procedure.

value type must be set AND "" is not a valid name.

This is very similar to defining a variable.

Set the value type:



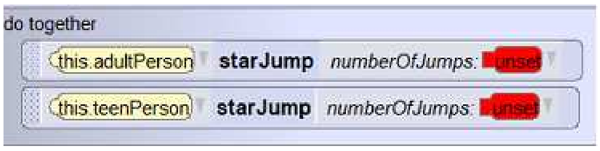
DecimalNumber	*examples:* 0.25, 1.0, 3.14, 98.6
WholeNumber	*examples:* 1, 2, 42, 100
Boolean	*examples:* true, false
TextString	*examples:* "hello", "goodbye"

For the starJump procedure, we would need a whole number.

Then give the parameter a meaningful name, for example numberOfJumps. For procedure parameters, you are not asked to give an initial value, because the value will be supplied when the procedure is called.

If the procedure has already been used in your program, you will need to tick the box next to 'I understand that I need to update the invocations to this procedure.' This is because you would have to next go back and change the lines where you had called this procedure to supply a parameter.

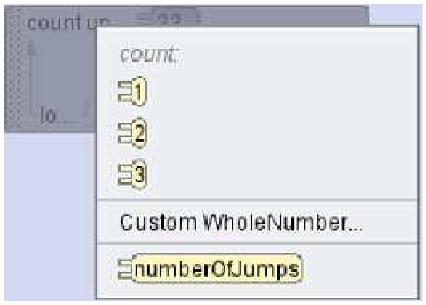
In the MyFirstMethod, this program called the starJump twice. Opening MyFirstMethod now shows a red flag next to each of them, showing the parameter is unset. To fix it, click the arrow at the end and select the number of jumps.



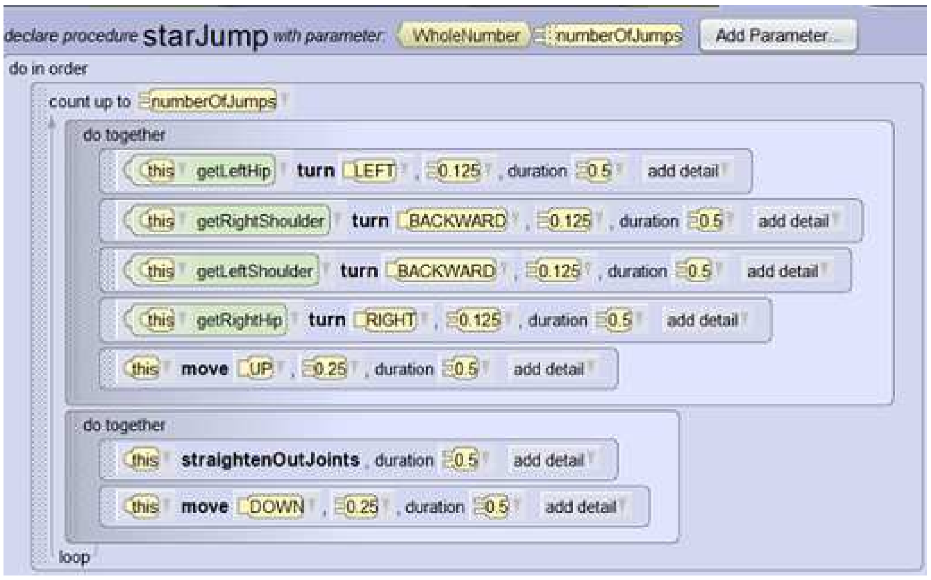
```
do together
    this.adultPerson  starJump  numberOfJumps: unset
    this.teenPerson  starJump  numberOfJumps: unset
```

Next, we would want to change the procedure to make use of this parameter.

If we drag in a count loop, we see that, along with the suggested numbers, we can choose numberOfJumps. This would repeat the jumps according to the supplied parameter.



Next, we'd drag the code into the count loop, so the finished procedure would look like this:



```
declare procedure starJump with parameter: WholeNumber numberOfJumps   Add Parameter...
do in order
    count up to numberOfJumps
        do together
            this getLeftHip turn LEFT , 0.125 , duration 0.5   add detail
            this getRightShoulder turn BACKWARD , 0.125 , duration 0.5   add detail
            this getLeftShoulder turn BACKWARD , 0.125 , duration 0.5   add detail
            this getRightHip turn RIGHT , 0.125 , duration 0.5   add detail
            this move UP , 0.25 , duration 0.5   add detail
        do together
            this straightenOutJoints , duration 0.5   add detail
            this move DOWN , 0.25 , duration 0.5   add detail
    loop
```

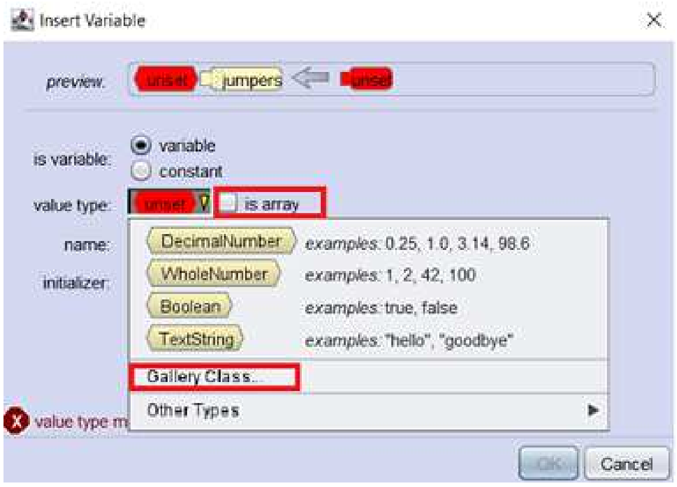
2.4 Using arrays

Arrays are lists, and they can be very useful with procedures. Suppose we had four bipeds in our program, and we wanted only three of them to do the star jumps,

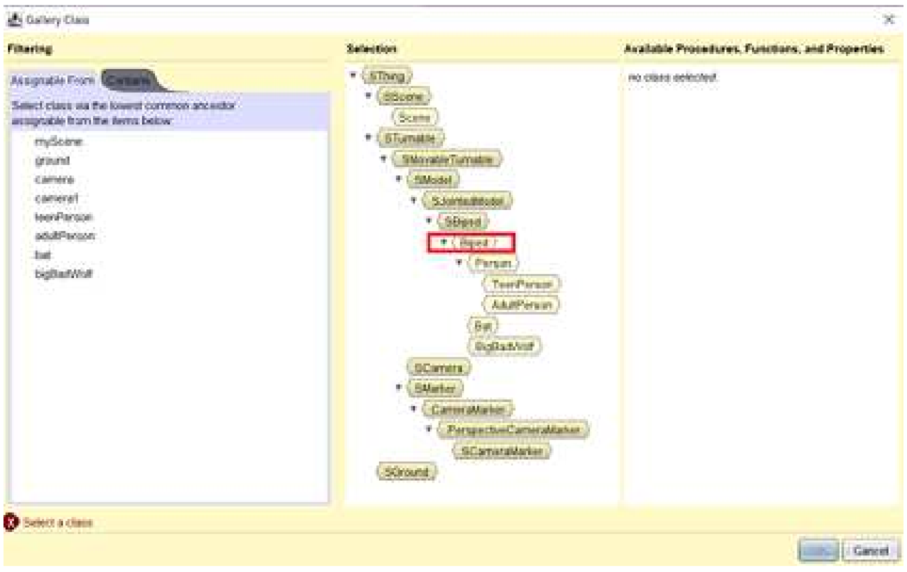
We can create an array, or list, of the bipeds who should do the jumps, and create a procedure that allows a group of bipeds to jump together.

In this example, the scene has these characters:

First we create an array of the objects that should do the jumps. An array is a variable, so in MyFirstMethod we first define the variable. From the code editor, when we drag in ‘Variable’, we see the familiar pop-up as below.

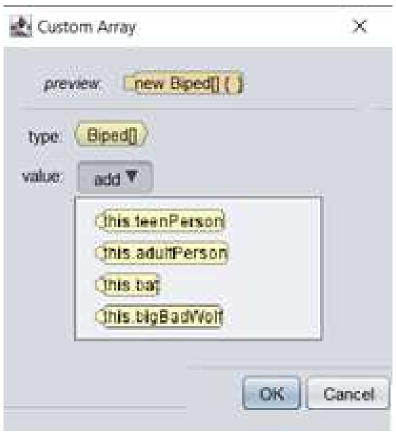


The ‘is array’ box, highlighted in the image, must be ticked. As highlighted with the red rectangle, choose ‘Gallery Class’ as the value type. This pops up the following screen.



Since the starJump procedure works with bipeds, the variable type should be Biped, so we choose this from the class hierarchy as shown in the image.

Next we need to set the initial value of the array. We click the initializer, and choose Custom Array. This pops up a window where we can add the bipeds that we want to place in the list.



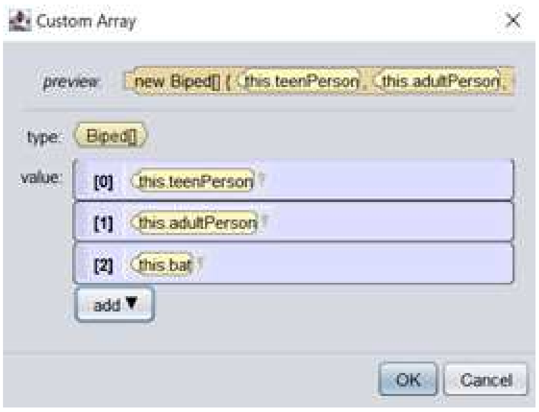
Custom Array

preview: new Biped[] { }

type: Biped[]

value: add ▼

For each object we want to add to the array, we click 'add', then choose the object from the list of available objects shown. Once done, the list should look like this:



Custom Array

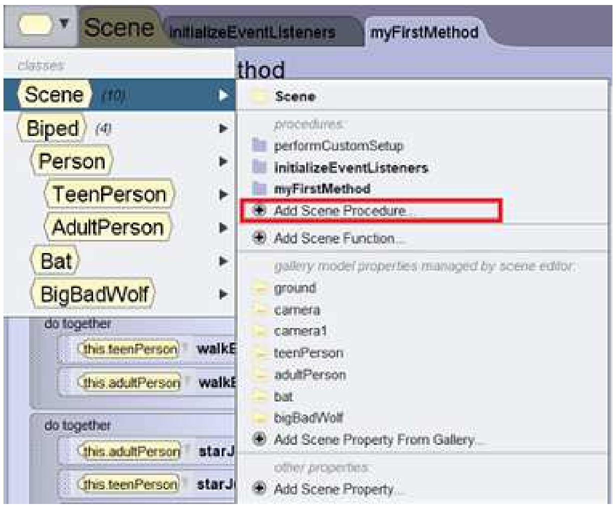
preview: new Biped[] { this.teenPerson, this.adultPerson,

type: Biped[]

value:
[0] this.teenPerson
[1] this.adultPerson
[2] this.bat

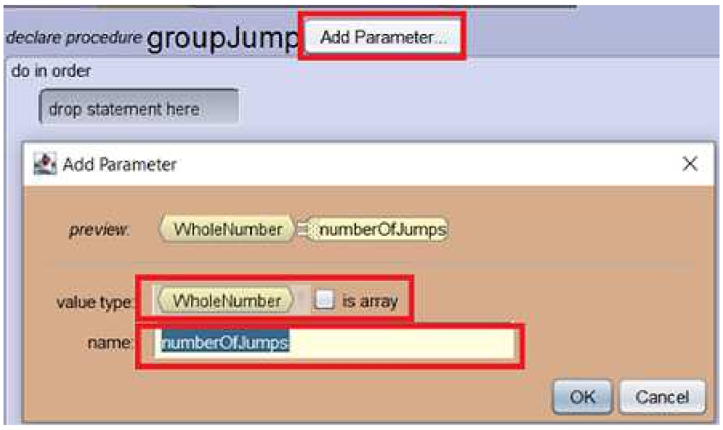
It's possible to remove an item from the list by clicking it, then clicking the cross in the right corner. When done, clicking 'Ok' will create the array.

Next we need to add a new procedure called groupJump. It doesn't really belong to the Biped class, because it's not something a single biped can do. Instead, we'll create it as belonging to our scene. In the code editor, click the navigator, choose Scene, then click 'Add Scene Procedure' as shown in the image.

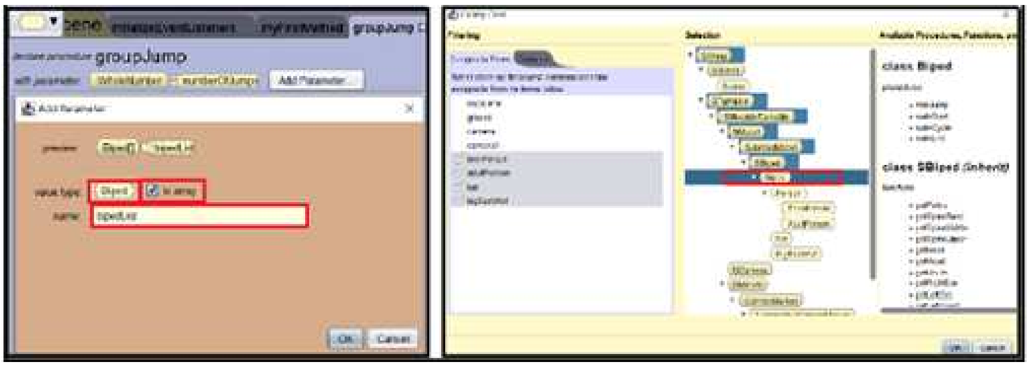


Name the procedure ‘groupJump.’ This will open a new tab for it in the code editor. We want the procedure to call the starJump procedure for each biped in the array. Since the starJump procedure needs a single parameter – the number of jumps to perform – this procedure will also need the same parameter. It will also need the array of bipeds who will jump as a parameter.

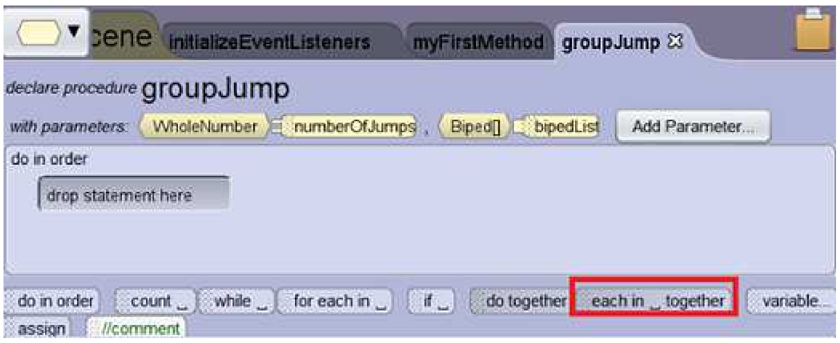
The first step is to add these parameters one at a time. Click ‘Add Parameter’ and fill in the details in the pop-up box as shown in the image. This creates numberOfJumps as the first parameter.



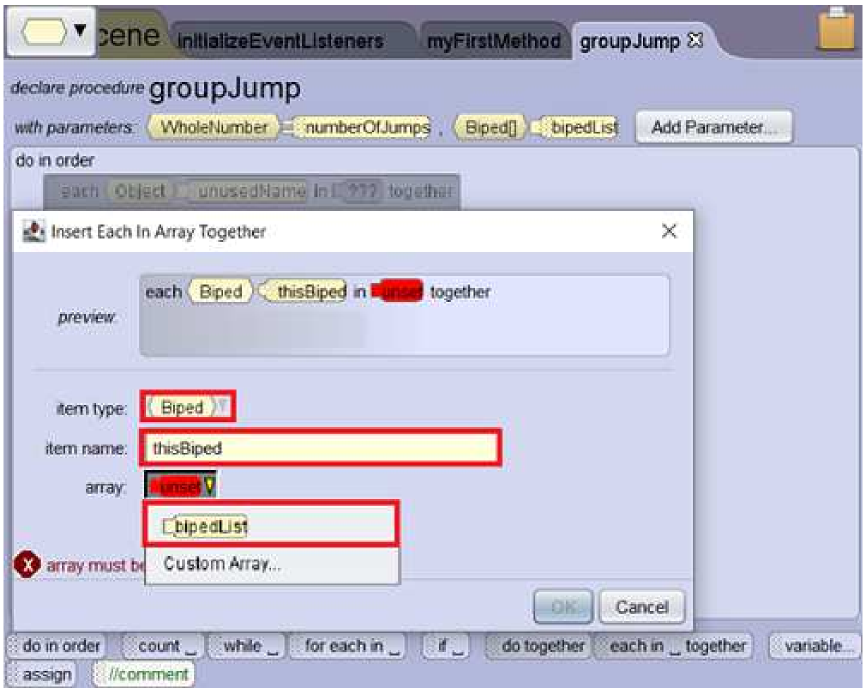
Next add the second parameter, which will be the array of bipeds who will jump. We’ll call it **bipedList.** Select ‘Add Parameter’. Set the value type to Gallery Type and then Biped. Tick ‘is array’, and name the parameter as per image below.



Now we’re ready to add the code. We want each biped in the array to do the star jumps at the same time. To do this, we can use a new programming construct: ‘each in together’, which does exactly what we want.



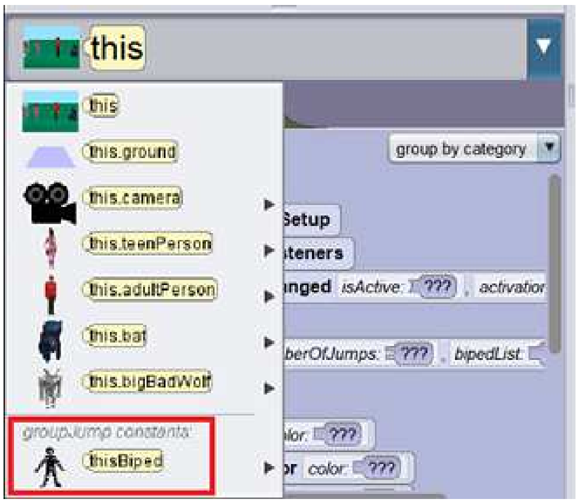
Drag it in from the bottom of the screen. This will pop up a window where you can fill in details of the array that it will use.



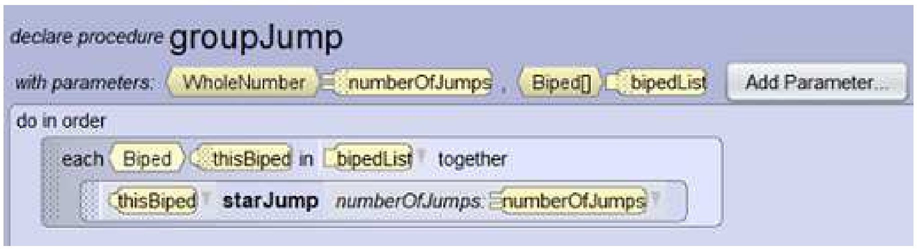
As shown in the image, set the item type to Biped (Gallery Class) and the name to **bipedList.** Clicking the arrow next to 'array' shows a list of arrays that you can choose from. One of the choices is **bipedList,** which is the parameter we defined for the procedure. This is the one we want. 'Custom Array' would only be used if you wanted a new array just for this action block.

The name we used here is the name we would use to refer to the biped from the array that is currently being worked with.

It can now be selected from the object selector, just like any other object.



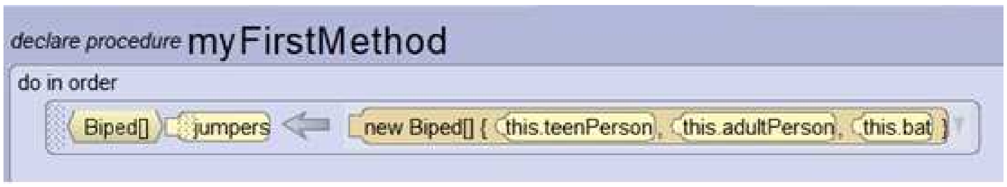
Since it's a biped, it will have the starJumps procedure, just like any other biped. Drag this procedure into the 'each...together' block, and use **numberOfJumps** (the first procedure parameter) as the parameter. The code should look like this:



```
declare procedure groupJump
with parameters: WholeNumber numberOfJumps , Biped[] bipedList   Add Parameter...
do in order
    each Biped thisBiped in bipedList together
        thisBiped starJump numberOfJumps: numberOfJumps
```

Now we need to change myFirstMethod to call this procedure.

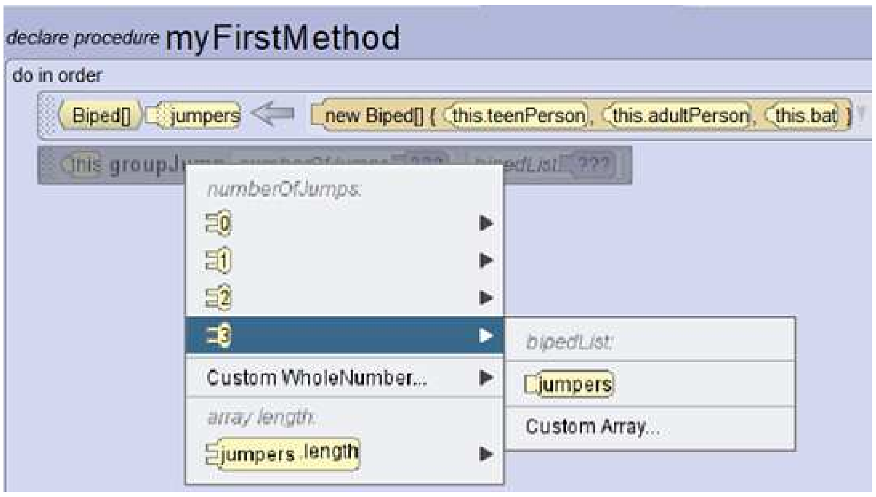
After we defined the array variable **jumpers**, myFirstMethod looked like this:



```
declare procedure myFirstMethod
do in order
    Biped[] jumpers ⇐ new Biped[] { this.teenPerson, this.adultPerson, this.bat }
```

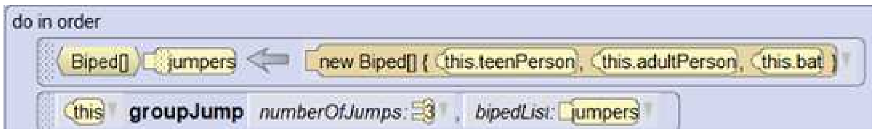
All we need to do now is to call our new procedure. Since the new procedure **groupJump** was created as belonging to the scene, we need to set the current object to 'this' from the object selector.

groupJump is listed as a procedure, and we can drag it in. Choose a number for the first parameter, and choose **jumpers** for the second parameter.



```
declare procedure myFirstMethod
do in order
    Biped[] jumpers ⇐ new Biped[] { this.teenPerson, this.adultPerson, this.bat }
    this groupJump numberOfJumps: ??? bipedList: ???
```

myFirstMethod now looks like this:



```
do in order
    Biped[] jumpers ⇐ new Biped[] { this.teenPerson, this.adultPerson, this.bat }
    this groupJump numberOfJumps: 3 , bipedList: jumpers
```

Now run the program. All the objects except the wolf should do three starjumps.

3 More on scene setting

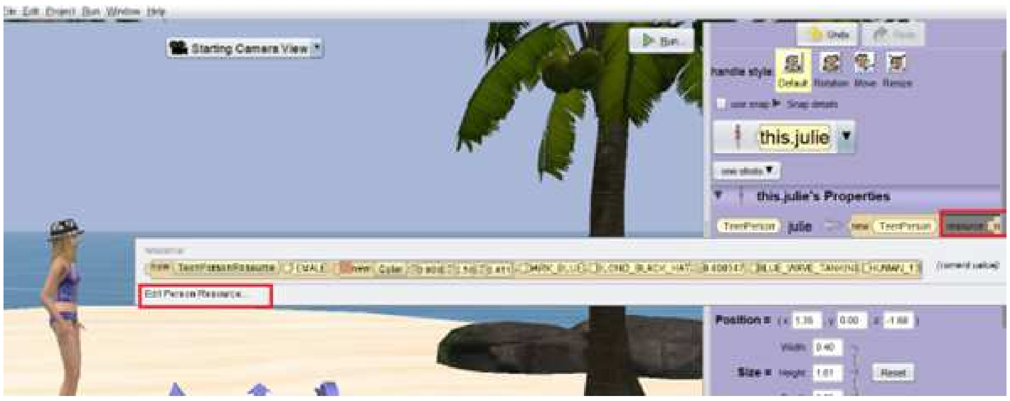
3.1 Using more than one scene in your world

Alice worlds have a single scene – the world, and there are several things that are common to the entire world – the type of ground (e.g. grass, sea, desert), the atmosphere colour (sky), lighting and fog density. These are all set when you first pick either a blank slate or a starter world for your project.

If your story or game needs a complete change of scene, you need to use a bit of ingenuity to achieve it.

Some hints:

- Since there is only one **ground** object, both scenes will have to be set up with the same type of ground in the scene editor. However, at the point in your program where the scene must be changed, you can use the **setPaint** to change the type of ground.
- You could also use **terrain** objects to cover the ground in one of your scenes (use the search in the gallery to find them.
- The methods of **this** (the scene) can be used at the point in the program where the scene switches, to change the atmosphere colour, lighting and fog density if needed.
- Set the two scenes up some distance away from each other (at least 100 difference in X position), and have a camera marker for each scene to quickly move the camera to the right position in your program.
- Use object markers to make it easy to move characters between scenes. Object markers can be set by selecting an object at the position where you want the marker, and then creating the marker. You will find object markers just above the camera markers. To move an object to a marker, use its moveTo method.
- You could also create clones of your characters, and have one clone in each scene, since you would often want the characters to be wearing different clothes. To create a clone, hold down the ALT key while using the move handle; this will create a second identical character. You can change the clone character's initial properties to change the way he is dressed by clicking 'resource' at the top of the properties panel as highlighted in the image, then click 'Edit person resources' as shown. This will pop up the screen that allows you to choose the clothes, the hat etc.



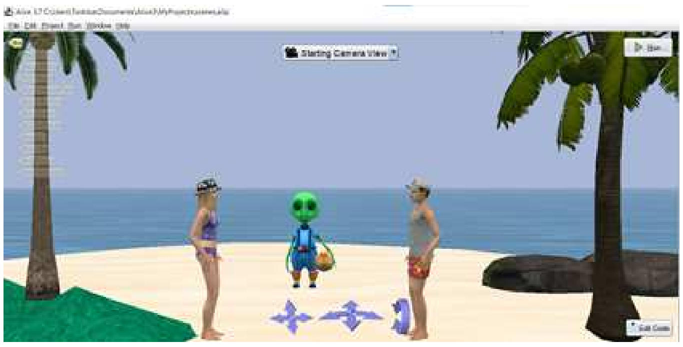
Example – moving from beach to moonscape.

Scenario:

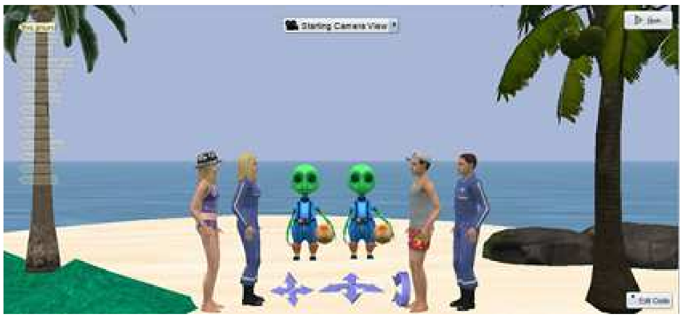
- Sam and Julie are enjoying a day on the beach.
- Their alien friend Klayck appears, and tells them the Kruges have stolen the radiobionic globulator, and will use it to destroy the galaxy. Tells them they are needed immediately.
- Klayck uses the spinulator to beam them up to Galactic Outpost 1
- They all appear at the outpost.

Steps to achieve this:

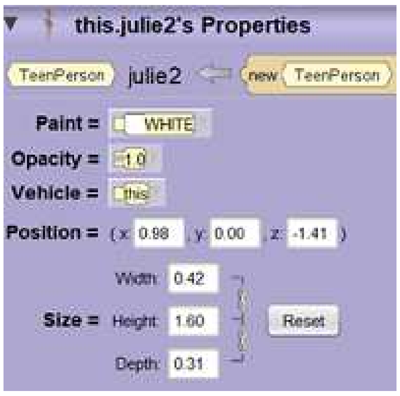
a) Create the beach scene as shown below, and set a camera marker



b) Create clones of all three characters, and also the spinulator. Edit Sam and Julie's resources to change their clothes, so that the scene looks like this:



c) Julie2's properties may look like this:



this.julie2's Properties

TeenPerson julie2 ⇐ new TeenPerson

Paint = WHITE

Opacity = 1.0

Vehicle = this

Position = (x: 0.98, y: 0.00, z: -1.41)

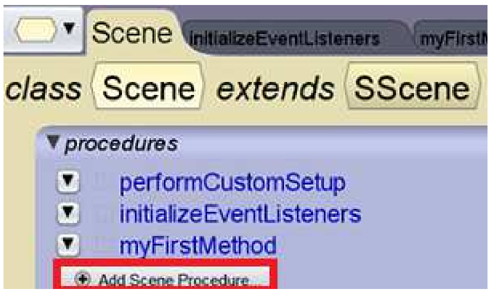
Size = Width: 0.42, Height: 1.60, Depth: 0.31

Reset

Change the x position property by adding 100 to make it 100.98. This will move her far away to the left, where we will set up our second scene. Do the same for sam2, klayck2 and spinulator2.

d) Change the camera properties to add 100 to its x position. This will move the camera to the position of the new scene. Set a camera marker.
e) Set up the second scene. Note that the ground is still set to Ocean, so it will look rather strange, but in the code, we will use setPaint to change it when the program reaches that point.

f) In the code editor, set up two new scene procedures by clicking Scene > Add Scene Procedure. They will be named changeToBeach and changeToSpaceOutpost



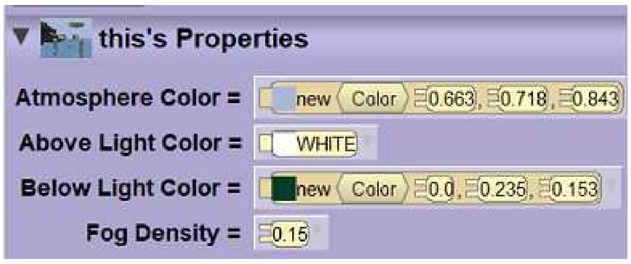
Scene | initializeEventListeners | myFirst

class Scene extends SScene

procedures
- performCustomSetup
- initializeEventListeners
- myFirstMethod

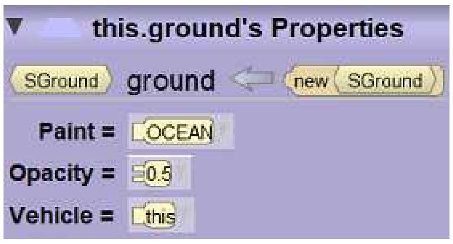
Add Scene Procedure...

g) Before we start coding, make a note of the properties of ‘**this**’ – the scene and **ground**. These are the properties that we will want to restore when we change back to the beach scene. At the moment, they are set as shown:



this's Properties

Atmosphere Color = new Color(0.663, 0.718, 0.843)
Above Light Color = WHITE
Below Light Color = new Color(0.0, 0.235, 0.153)
Fog Density = 0.15



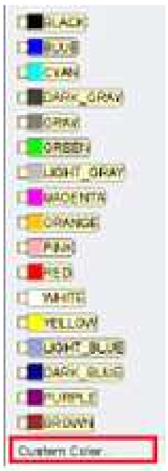
this.ground's Properties

SGround ground ⇐ new SGround()

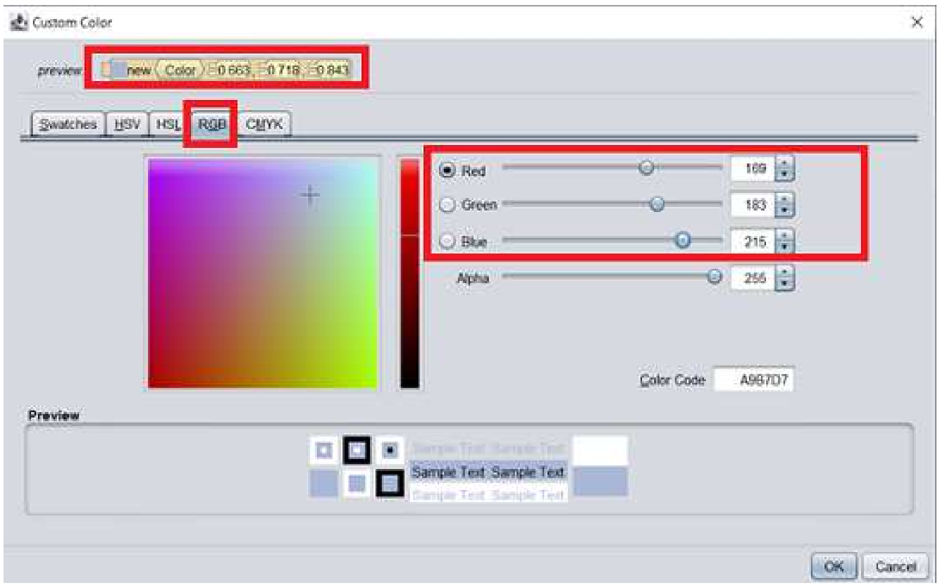
Paint = OCEAN
Opacity = 0.5
Vehicle = this

h) The code for **changeToBeach** will need to therefore set:
- The atmosphere colour
- The above light colour
- The below light colour
- The fog density
- The paint and opacity of the ground.

Since the atmosphere and below light colours are not standard, they will need to be changed using Custom Color.



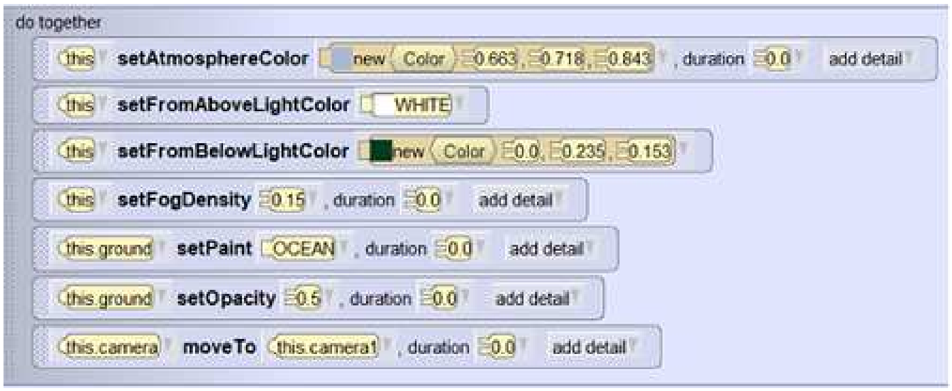
The colour properties shown in the properties panel for atmosphere colour (0.663,0.718,0.843) indicate the amounts of the three primary colours present – red, green and blue. These are usually known as RGB settings. On clicking 'Custom Color', it shows a colour selector pop-up. Since we want to set the RGB to the original settings, click the RGB tab, then move each of the sliders until the setting at the top shows the required RGB combination.



preview: new Color(0.663, 0.718, 0.843)

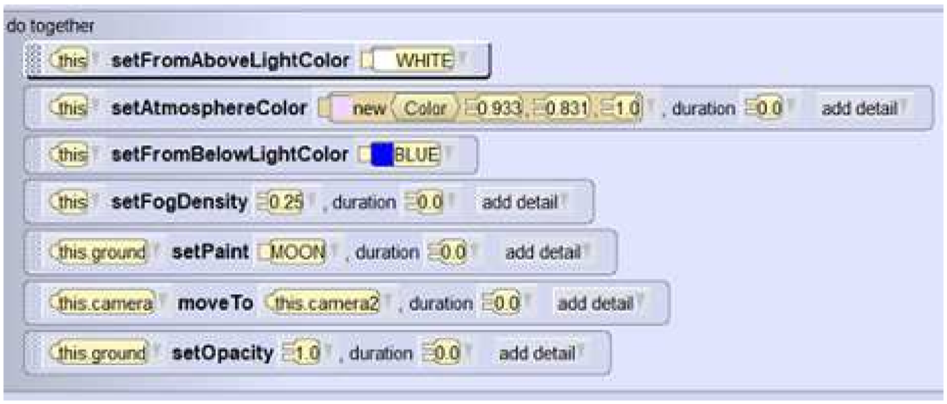
Red 169
Green 183
Blue 215
Alpha 255
Color Code A9B7D7

The code for changeToBeach will look like this:



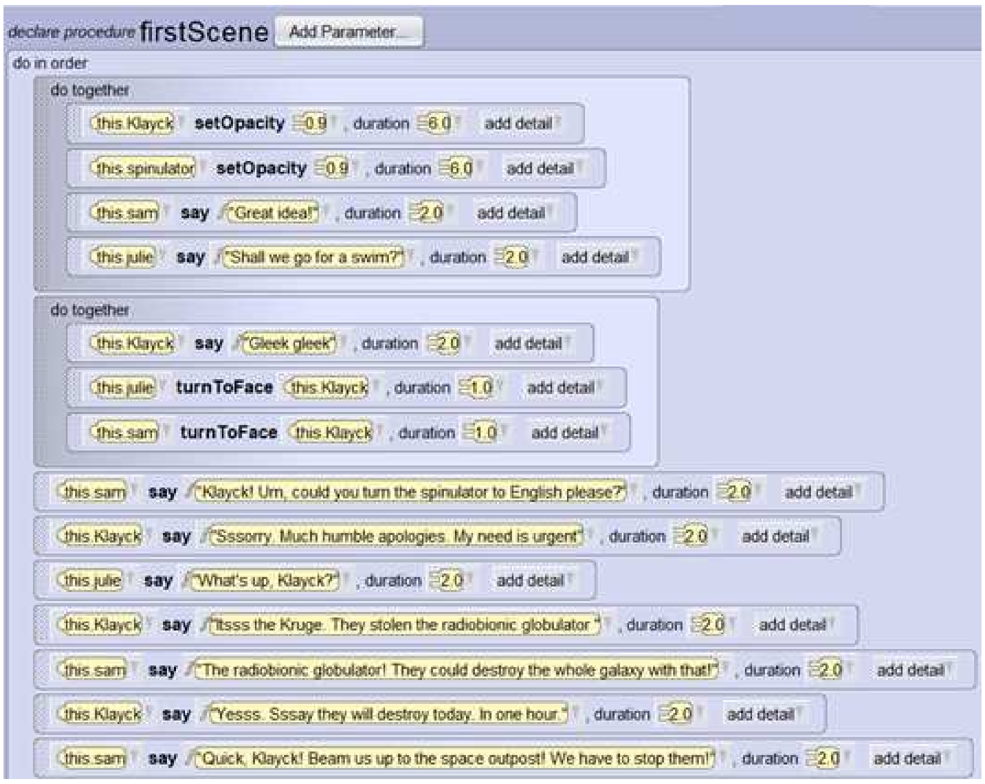
```
do together
    this setAtmosphereColor new Color( 0.663, 0.718, 0.843) , duration 0.0  add detail
    this setFromAboveLightColor WHITE
    this setFromBelowLightColor new Color( 0.0, 0.235, 0.153)
    this setFogDensity 0.15 , duration 0.0  add detail
    this.ground setPaint OCEAN , duration 0.0  add detail
    this.ground setOpacity 0.5 , duration 0.0  add detail
    this.camera moveTo this.camera1 , duration 0.0  add detail
```

i) The code for **changeToSpaceOutpost** will use the same set procedures as **changeToBeach**, so we can use the clipboard to copy the code across, then change it to suitable values. It might look like this:

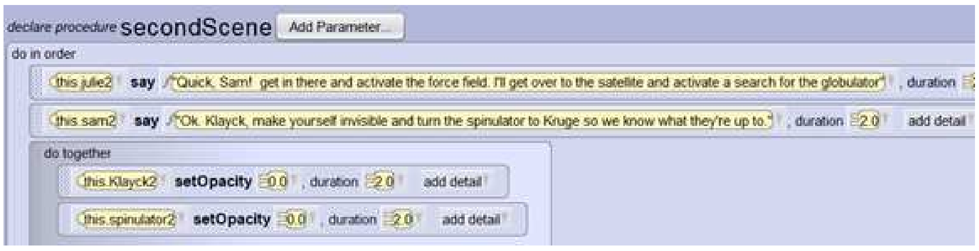


```
do together
    this setFromAboveLightColor WHITE
    this setAtmosphereColor new Color 0.933, 0.831, 1.0 , duration 0.0   add detail
    this setFromBelowLightColor BLUE
    this setFogDensity 0.25 , duration 0.0   add detail
    this.ground setPaint MOON , duration 0.0   add detail
    this.camera moveTo this.camera2 , duration 0.0   add detail
    this.ground setOpacity 1.0 , duration 0.0   add detail
```

j) Put the code needed for the beach scene and for the space outpost scene into two methods, firstScene and secondScene:

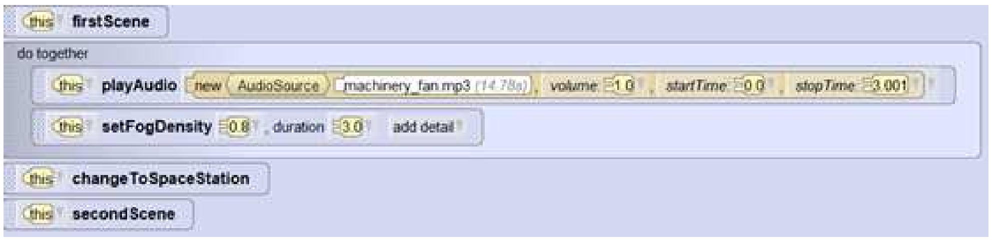


```
declare procedure firstScene  Add Parameter...
do in order
    do together
        this.Klayck setOpacity 0.9 , duration 6.0   add detail
        this.spinulator setOpacity 0.9 , duration 6.0   add detail
        this.sam say "Great idea!" , duration 2.0   add detail
        this.julie say "Shall we go for a swim?" , duration 2.0   add detail
    do together
        this.Klayck say "Gleek gleek" , duration 2.0   add detail
        this.julie turnToFace this.Klayck , duration 1.0   add detail
        this.sam turnToFace this.Klayck , duration 1.0   add detail
    this.sam say "Klayck! Um, could you turn the spinulator to English please?" , duration 2.0   add detail
    this.Klayck say "Sssorry. Much humble apologies. My need is urgent" , duration 2.0   add detail
    this.julie say "What's up, Klayck?" , duration 2.0   add detail
    this.Klayck say "Itsss the Kruge. They stolen the radiobionic globulator " , duration 2.0   add detail
    this.sam say "The radiobionic globulator! They could destroy the whole galaxy with that!" , duration 2.0   add detail
    this.Klayck say "Yesss. Sssay they will destroy today. In one hour." , duration 2.0   add detail
    this.sam say "Quick, Klayck! Beam us up to the space outpost! We have to stop them!" , duration 2.0   add detail
```



```
declare procedure secondScene  Add Parameter...
do in order
    this.julie2 say "Quick, Sam!  get in there and activate the force field. I'll get over to the satellite and activate a search for the globulator" , duration
    this.sam2 say "Ok. Klayck, make yourself invisible and turn the spinulator to Kruge so we know what they're up to." , duration 2.0   add detail
    do together
        this.Klayck2 setOpacity 0.0 , duration 2.0   add detail
        this.spinulator2 setOpacity 0.0 , duration 2.0   add detail
```

k) MyFirstMethod will then look like this:



```
this firstScene
do together
    this playAudio new AudioSource machinery_fan.mp3 (14.78s) , volume 1.0 , startTime 0.0 , stopTime 3.001
    this setFogDensity 0.8 , duration 3.0    add detail
this changeToSpaceStation
this secondScene
```

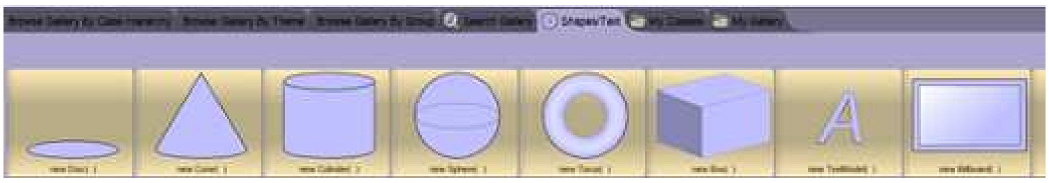
When the program is run, the beach scene will look like this:

And the space outpost scene like this:

The program can now easily switch between the two as the story progresses.

Using Shapes and Text

With a little imagination, the classes under the Shapes/Text tab of the gallery can be used for many things.

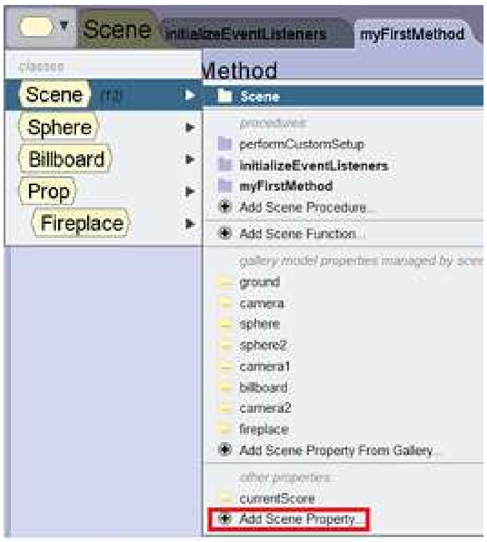


By setting their size and colours, and perhaps putting them together, you can turn them into all kinds of useful things. The 'Vehicle' property can be used to make sure two objects always move together. It's best to learn how to use them by experimenting and trying different things. Here's a few suggestions.

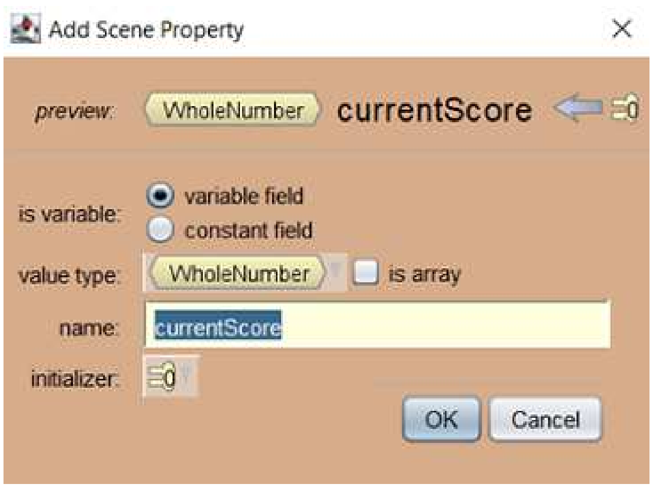
Billboards: These are flat, in other words they have height and width but no depth. They can be used as the background to a scoreboard, walls, ceilings and many other things. It's also possible to use the 'Front Paint' or 'Back Paint' property to set an image on the billboard. In the example below, I've set the 'Front Paint' image to a picture of a stone wall. I've placed a **Fireplace** object in front of the billboard.

TextModel: These are often used to display the score in a game, but could be used for many other things, for example a signboard. For the text to appear clearly, it's often a good idea to a text model in front of a billboard for contrast.

To keep score, you would need a variable, usually created as a property in the Scene object (all properties are simply variables that belong to a class rather than to a method) so that it can be accessed from any method. Add to it every time the gamer deserves more score, using an assignment. You can then use the setValue method of the text model to display the score.



Adding a Scene Property


Add Scene Property

preview: WholeNumber currentScore ⇐ 0

is variable: ● variable field ○ constant field

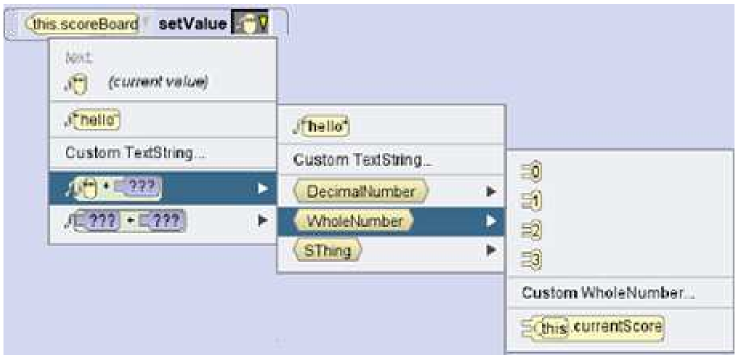
value type: WholeNumber ☐ is array

name: currentScore

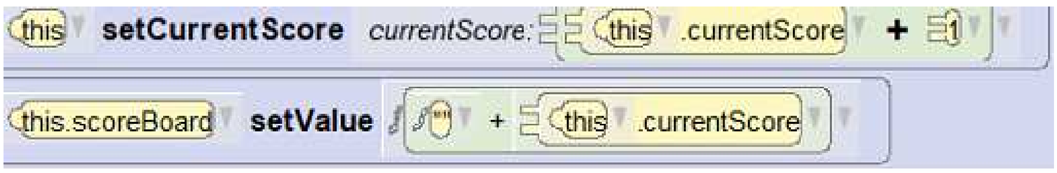
initializer: 0

To create the scoreboard, create a billboard and set it to the size you want. Create a textModel. If you don’t give it an initial value, you won’t be able to see it, so set it to NIL and set the paint to black so it shows up against the billboard. Name it scoreboard, and set it to a good size to fit the billboard. Use One shots to place it in front of the billboard, then adjust its position so it’s centred on the billboard. Set its vehicle to the billboard so they both move together.

When the gamer scores, you can then use the setValue method of the **scoreBoard** to display his score. Since setValue requires a parameter of data type ‘text’, you can’t do this directly. Alice won’t allow a variable to be set to a value that has a different data type. You can achieve it by putting together an empty text value together with the contents of the numeric score variable as shown in the image below.



Your scoring code may look like this:


```
this setCurrentScore currentScore: this.currentScore + 1
this.scoreBoard setValue "" + this.currentScore
```

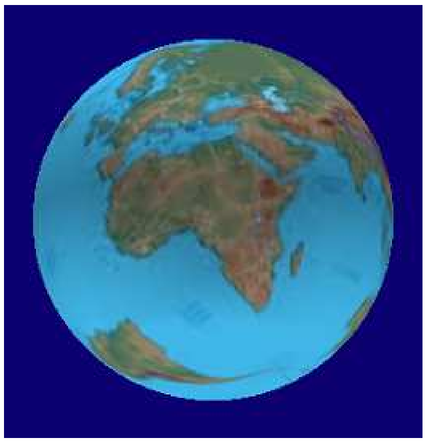
Note that since currentScore was defined as a scene property, the scene now has a method named **setCurrentScore** to set its value.

Using a sphere as a planet for a space game or story:

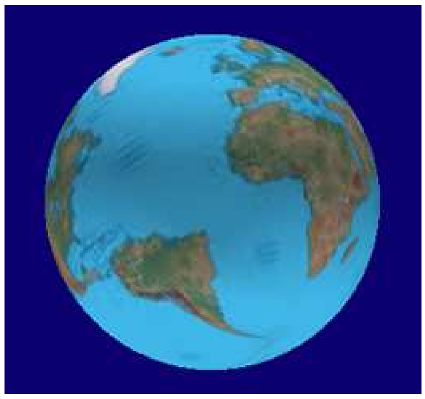
Like the billboard, the paint property of a sphere can be set to an image. However, Alice wraps the image around the sphere, stretching it as needed.

If I have an image named ‘map’ containing a world map like this:

and set the **paint** property of a sphere to this image, my image object looks like this:



and when rotated:



I can do the same thing with a picture of Jupiter:

to get this:

Let your imagination go wild!

4 Using Audacity to create sounds

Audacity is not part of Alice, and we will only be having a brief look at it to learn enough to make customised sound effects and voice-overs to use with Alice.

4.1.1 Sound files

There are many different file formats used for sound, for example .MP3 and .WAV. Alice can only work with sound files in .WAV format.

When you record and save in Audacity, it stores it in a project file in its own format. To use the recording in Alice, you would need to export the project to a file in .WAV format.

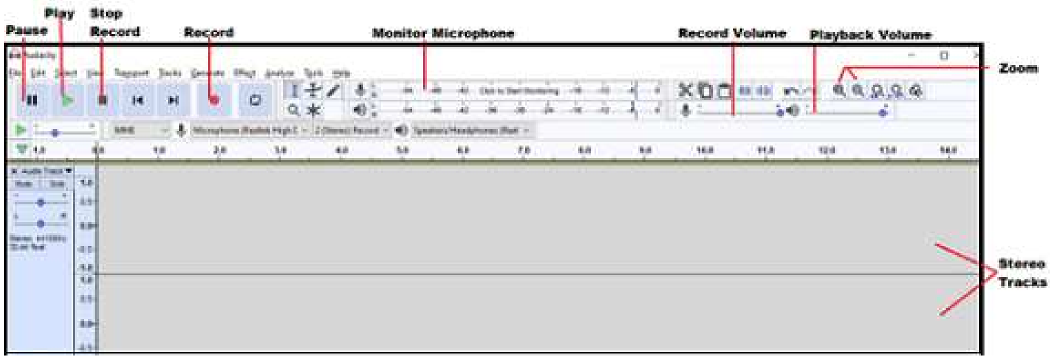
If you have a sound file that's not in .WAV format, and you want to use it in Alice, you can import it into an Audacity project and export it in .WAV format.

The Alice sound galleries are in this folder: C:\Program Files\Alice 3\application\sound-gallery. If you create a new folder there and use it for your exported audio files from Audacity, it will make your recordings easy to find in Alice.

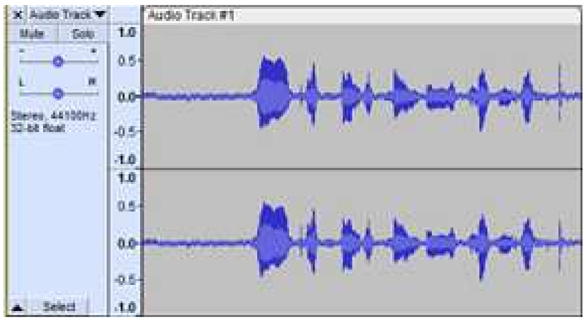
Once you have made a recording and exported it in .WAV format, you can use it with the playAudio method of any object.

4.1.2 The Audacity Screen

When you first open Audacity, the screen will look similar to the image below.



The stereo tracks will be empty until you either import an audio file, or make a recording. They would then show a graph of the audio as shown in the image below.



4.1.3 Recording sound

To make a recording, first click the 'Monitor Microphone' section to check that the mic is working and the recording level is not too high. When you're ready to record, click the red 'record' button. When you've finished, click the black 'stop' button. Click the playback button to hear the recording. To save it as an Audacity project, click 'File' from the menu, and save. To save it in audio format for use with Alice, click File, then Export, then Export as .WAV.

4.1.4 Editing the recording

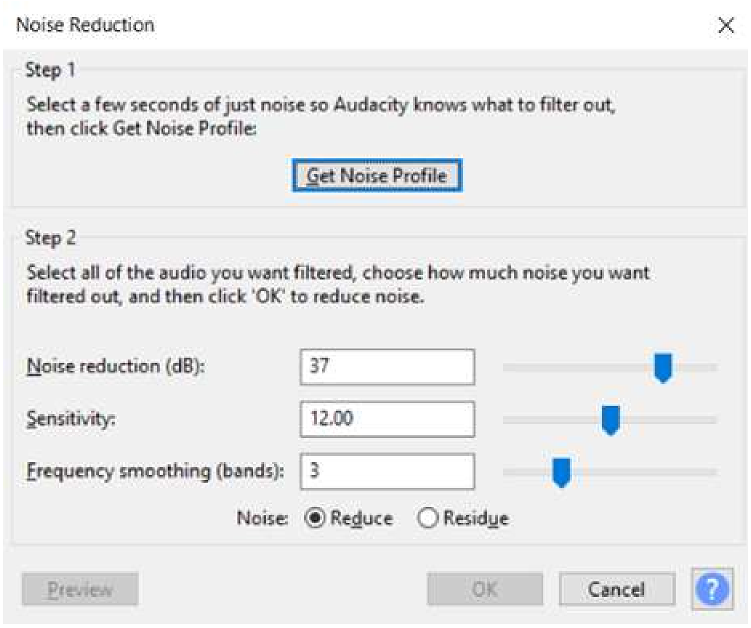
You will usually want to make some changes to the recording before exporting it as .WAV. Here are some common edits that may be needed. Some edits may affect the full audio, and others only a part of the audio. To select part of the audio, click inside the stereo track at the start of the part that needs to be changed, then drag the mouse across to the end of the part. To select the whole audio, press CTL A. At any time, the last edit can be undone using CTL Z.

4.1.4.1 Deleting

You will often need to cut silences from the beginning and end of the recording, and sometimes in the middle. You can do this by selecting the area and pressing the delete key.

4.1.4.2 Reducing noise

Background noise , such as the hum of the computer's fan, can spoil your recording. You can remove it by using the Noise Reduction effect. This is done in two steps. First, select an area of the recording that contains only noise and nothing else. Select Effects from the menu, then Noise Reduction. This pops up the following box:


Noise Reduction

Step 1

Select a few seconds of just noise so Audacity knows what to filter out, then click Get Noise Profile:

Step 2

Select all of the audio you want filtered, choose how much noise you want filtered out, and then click 'OK' to reduce noise.

Noise reduction (dB): 37

Sensitivity: 12.00

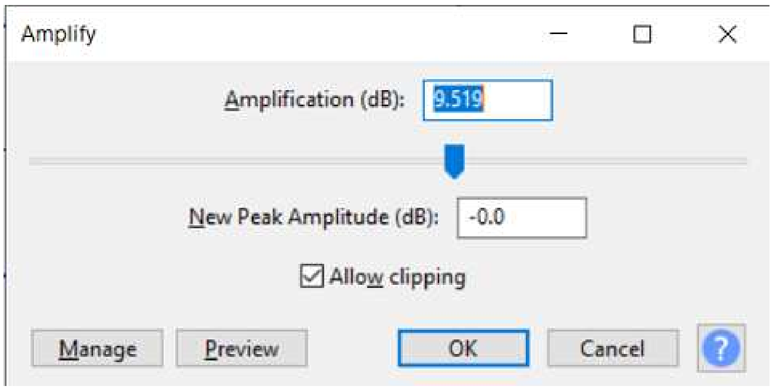
Frequency smoothing (bands): 3

Noise: (●) Reduce ( ) Residue

Click 'Get Noise Profile'. Next, select the whole recording (CTL A) and again select Effects > Noise Reduction. This time, click OK. The noise will be removed from the recording.

4.1.4.3 Increase or decrease the volume

Select the section where the volume should be changed, or select the whole recording. From the menu, click Amplify. The box shown below will pop up.


Amplify

Amplification (dB): 9.519

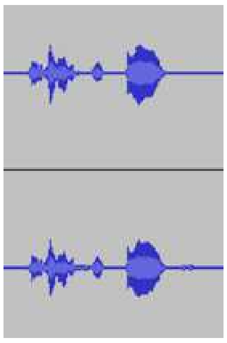
New Peak Amplitude (dB): -0.0

☑ Allow clipping

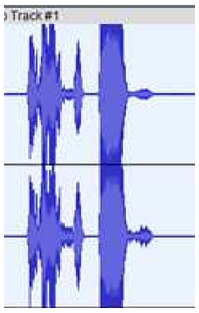
You can change the volume by dragging the slider up or down, then click Preview to make sure you now have the volume you want before clicking OK.

Be careful not to increase the volume too much, or you may find the audio becomes distorted because it has gone over the max and been chopped off, as per the example below.

Original audio:



Audio after volume has been increased too much:



4.1.4.4 *Change the sound of your voice*

If you are doing voice-overs for more than one character, you may want to change the sound of your voice for each character. You can experiment with some of the effects. Useful effects include:

- Change Pitch
- Change Speed
- Change Tempo
- Paulstretch
- Wahwah

5 Event handling

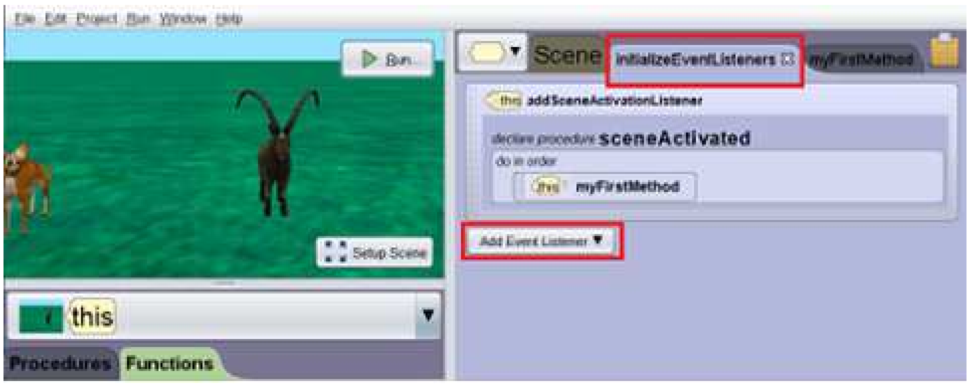
In programming, as in the real world, an event is something that happens. A program is said to be event-driven if it carries out some tasks based on an outside event. A method that is activated by an event is known as an event handler, or event listener. In Alice, there are five types of event that can activate an event handler:

• Scene Activation Event:	Occurs when the program is first started
• Timer Event:	Occurs when a given amount of time has elapsed
• Keyboard Event:	Occurs when the user presses a key on the keyboard
• Mouse Event:	Occurs when the mouse is used
• Position/Orientation Event:	Triggered by the position of an object

Uses of event handlers include:

- Detecting keystrokes or mouse movement to allow a user to control a game
- Detecting when two objects collide
- Popping up collectable items in a game at regular intervals

Event listeners are defined in the InitializeEventListeners tab in the code editor. New event listeners are created by clicking 'Add Event Listener'.



Since this tab can very quickly become too cluttered to work with, it's best to put the actual event handling code into another new procedure, and simply have the event listener call that procedure.

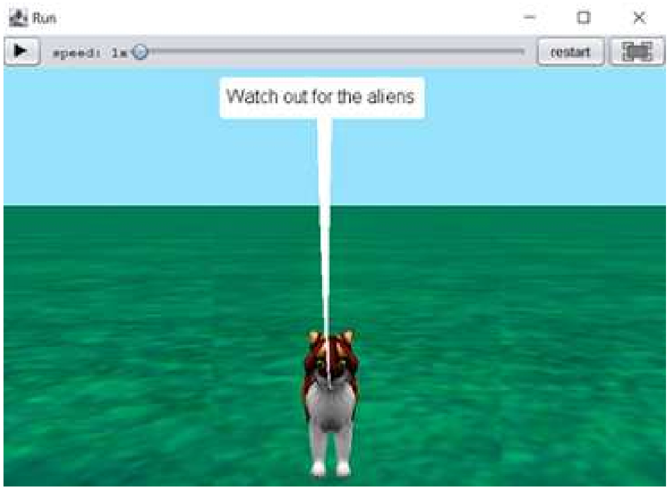
Note that once you include event handlers, your program will be doing more than one thing at the same time. Whatever code is put into **MyFirstMethod** will continue to run alongside the code in the event handler. If a second event is fired before the first event handler code has finished running, this will also run at the same time. You need to be aware of this, as it's possible for them to interfere with each other. For example, if **MyFirstmethod** sets a scene property, and the event handler also sets it, it's not always possible to be sure what its value will be.

5.1 Scene Activation Listener

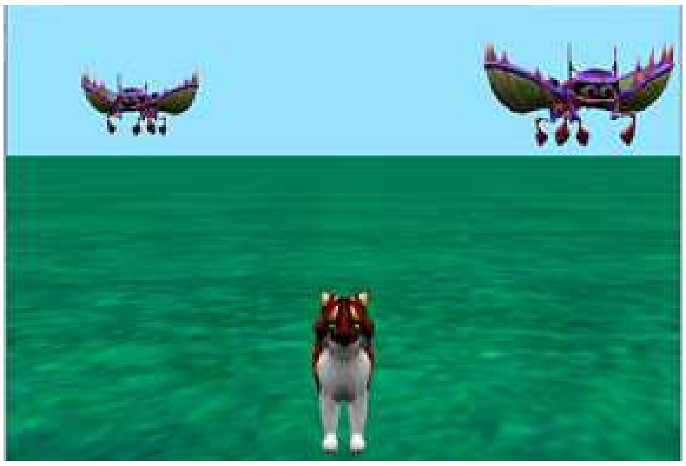
Looking at the image above, you can see there is already a method named **sceneActivated**. It has one line of code in it: a call to the **myFirstMethod** procedure. If you had any actions that you wanted carried out before calling **MyFirstMethod**, you would put them here. One example might be that you wanted a sound track for your program. You could change this method to put **myFirstMethod** and a **playAudio** into a 'do together' block in this procedure.

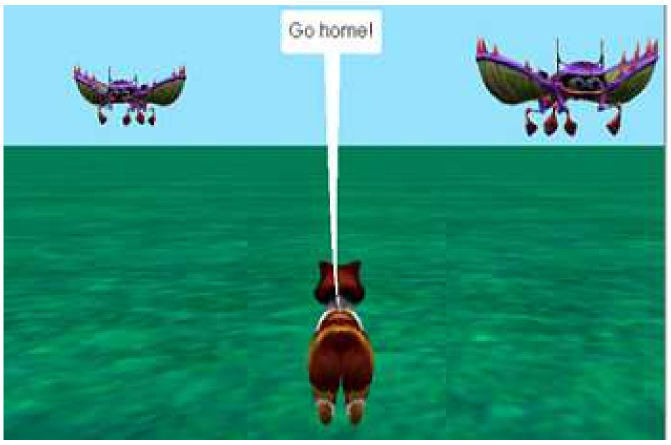
5.2 Timer Events

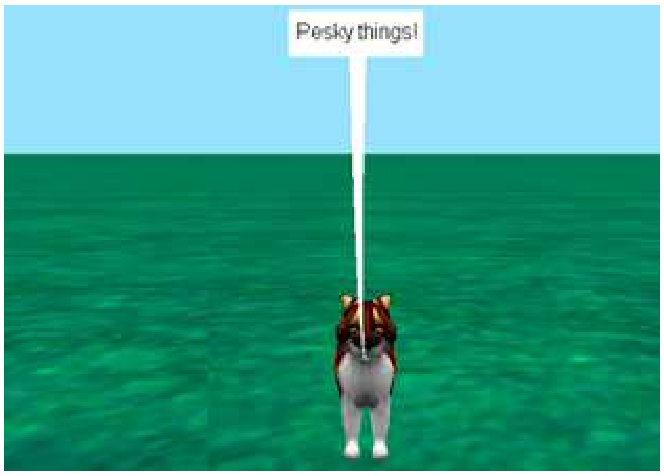
Timer events will fire whenever the time since the last fire (or since scene activation) reaches a given value. Let's look at an example where a manx cat is annoyed by alien robots every six seconds. In our scene, we have the manx cat and two alien robots. The alien robots have opacity set to 0 at the start. The timer event will make them visible, and the manx cat will turn around and tell them to go home. When running, the program will have the following sequence:



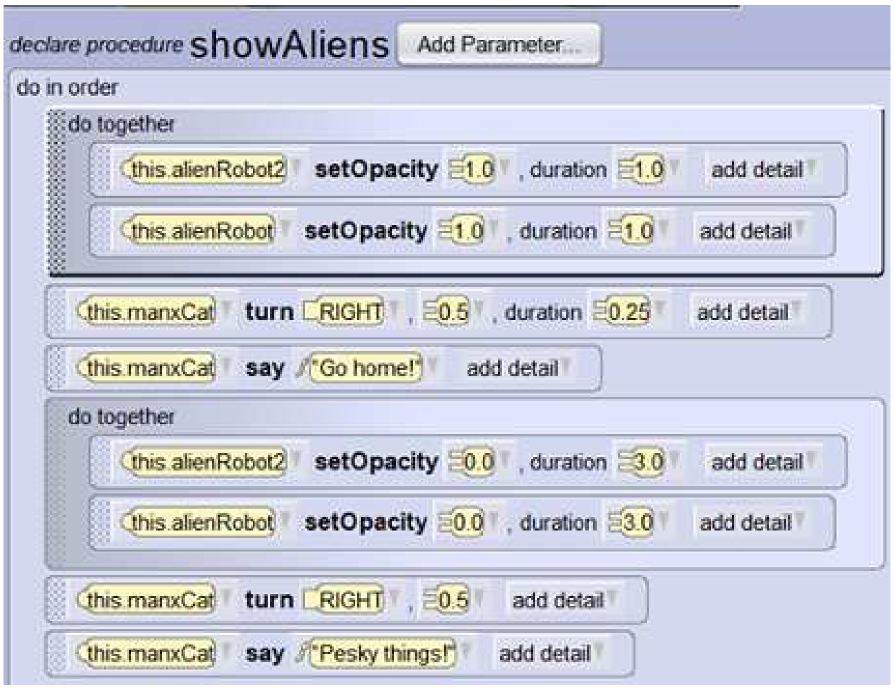
Every 6 seconds, the timer event will cause the action as below to happen:





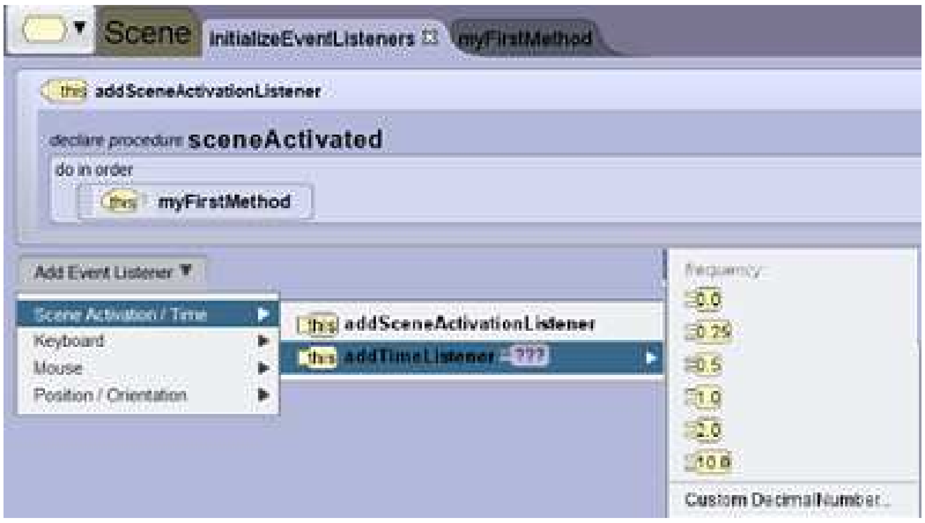


To achieve this, we first put the event handler code into a new procedure under Scene, which looks like this:



```
declare procedure showAliens  Add Parameter...
do in order
    do together
        this.alienRobot2  setOpacity 1.0 , duration 1.0  add detail
        this.alienRobot  setOpacity 1.0 , duration 1.0  add detail
    this.manxCat  turn RIGHT , 0.5 , duration 0.25  add detail
    this.manxCat  say "Go home!"  add detail
    do together
        this.alienRobot2  setOpacity 0.0 , duration 3.0  add detail
        this.alienRobot  setOpacity 0.0 , duration 3.0  add detail
    this.manxCat  turn RIGHT , 0.5  add detail
    this.manxCat  say "Pesky things!"  add detail
```

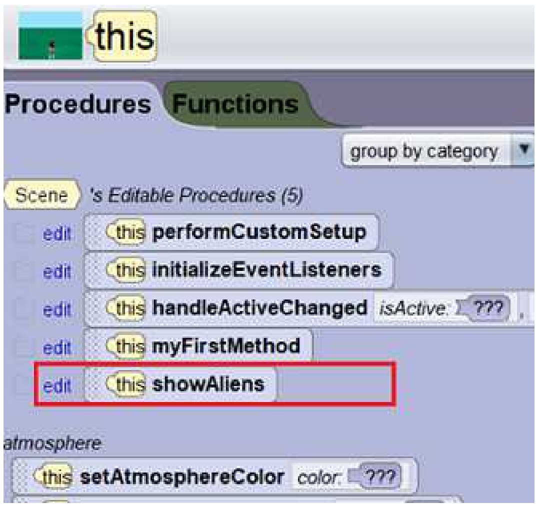
Next we add the event listener. Working in the InitializeEventListeners tab of the code editor, click 'Add Event Listener.' This pops up the following:



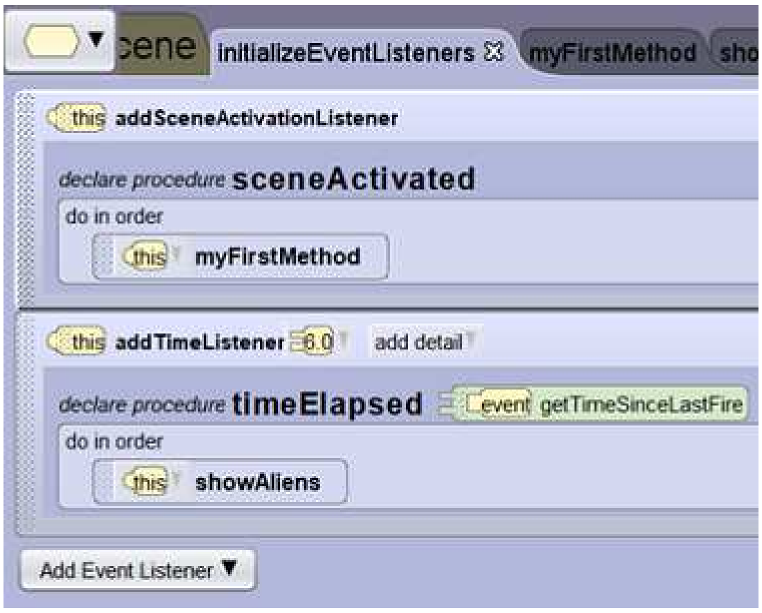
```
this addSceneActivationListener
    declare procedure sceneActivated
    do in order
        this myFirstMethod
```

First choose which type of event listener; this would be 'Scene Activation/Timer'. Then choose 'addTimeListener', and choose how often the listener will be activated.

This will create a new method in this tab named timeElapsed. Within this method, place a single instruction to call the showAliens procedure created in the last step. Since showAliens was created as a method of the scene, it will be listed as one of the scene's procedures, and can be dragged into the code.

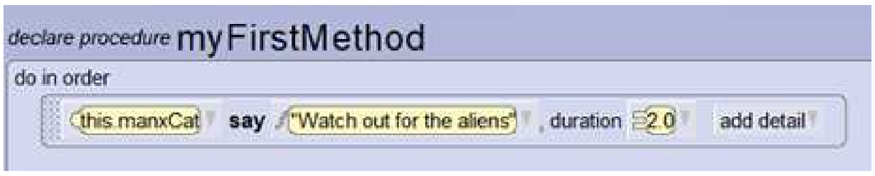

this
Procedures
Functions
group by category
Scene 's Editable Procedures (5)
edit this performCustomSetup
edit this initializeEventListeners
edit this handleActiveChanged isActive: ???
edit this myFirstMethod
edit this showAliens
atmosphere
this setAtmosphereColor color: ???

The InitializeEventListeners tab will now look like this:



```
this addSceneActivationListener
  declare procedure sceneActivated
    do in order
      this myFirstMethod
this addTimeListener 6.0 add detail
  declare procedure timeElapsed event getTimeSinceLastFire
    do in order
      this showAliens
```

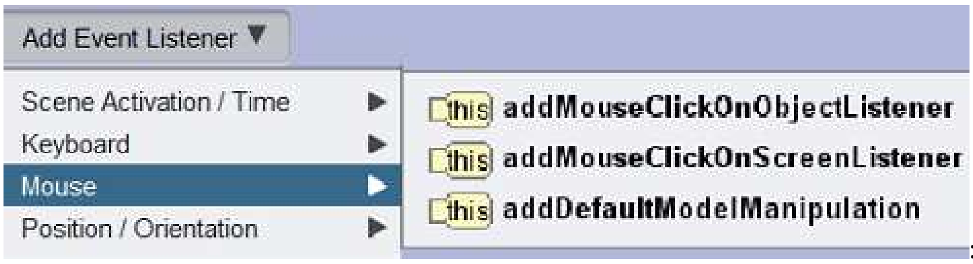
MyFirstMethod is not affected by any of this; it runs idependently. As usual, it contains whatever code needs to run when the program is started.



```
declare procedure myFirstMethod
  do in order
    this.manxCat say "Watch out for the aliens" , duration 2.0 add detail
```

5.3 Mouse Events

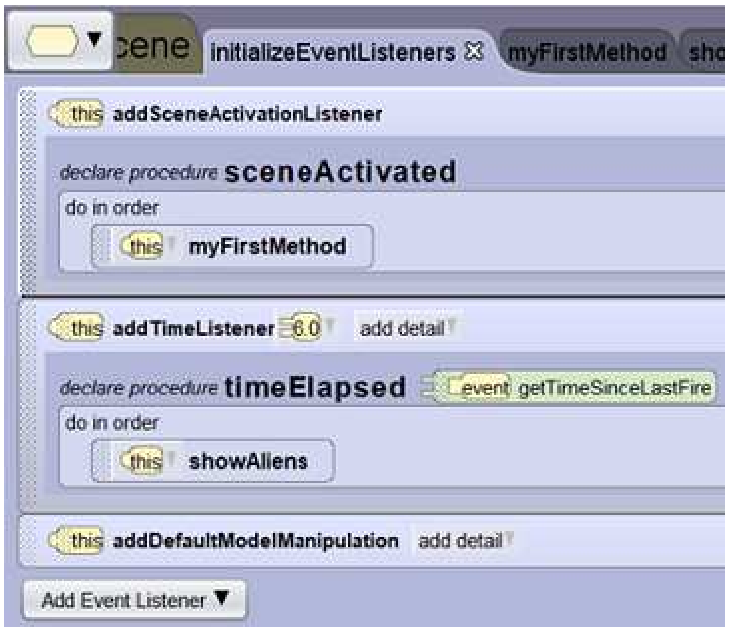
Mouse event listeners allow the program to detect mouse activity. There are a few different types of mouse listeners, as you will see if you select Mouse when adding an event listener:


Add Event Listener
Scene Activation / Time
Keyboard
Mouse
Position / Orientation
this addMouseClickOnObjectListener
this addMouseClickOnScreenListener
this addDefaultModelManipulation
:

5.3.1 addDefaultModelManipulation

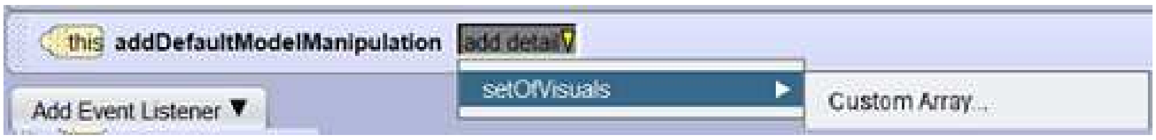
This is the simplest form of mouse listener. It does not allow any code to be added to the method; it simply activates the Alice default, which allows the user to drag any object on the screen to a new location with the mouse. The object can be dragged left, right, forward and backward. By holding down the shift key when dragging, it can also be moved up and down.

Choosing it gives the following in the 'Initialize Event Listeners' tab:

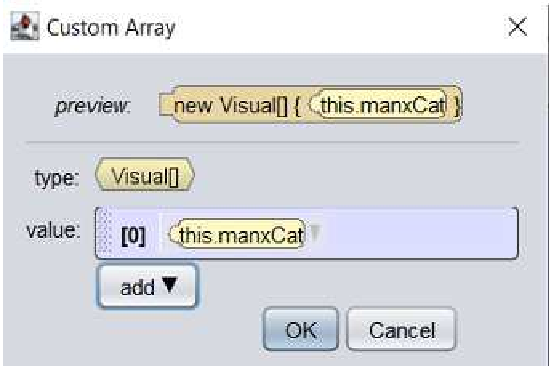


```
this addSceneActivationListener
  declare procedure sceneActivated
    do in order
      this myFirstMethod
this addTimeListener 6.0  add detail
  declare procedure timeElapsed  event getTimeSinceLastFire
    do in order
      this showAliens
this addDefaultModelManipulation  add detail
```

As it stands, there are disadvantages to using it: the user can drag any object around the screen, including your scenery. Once added, you can modify it so that only a limited set of one or more objects can be dragged. To do this, click the arrow next to 'add details', and choose 'setOfVisuals' > 'Custom array'.



You can now create an array of objects that the user will be allowed to control with the mouse. Often, there would be only one: the main character in the story or game.

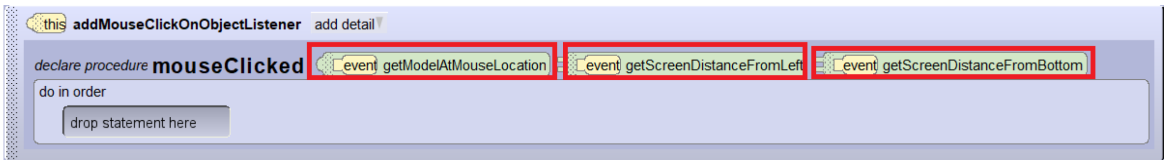


In this case, I have only added the Manx cat.

5.3.2 addMouseClickOnObjectListener

This listener will be triggered whenever the user clicks on any object on the screen.

Adding this listener gives the following method. Note the items highlighted in red: these are the parameters to the method, and they can be used within the method to get more information about the object that was clicked.



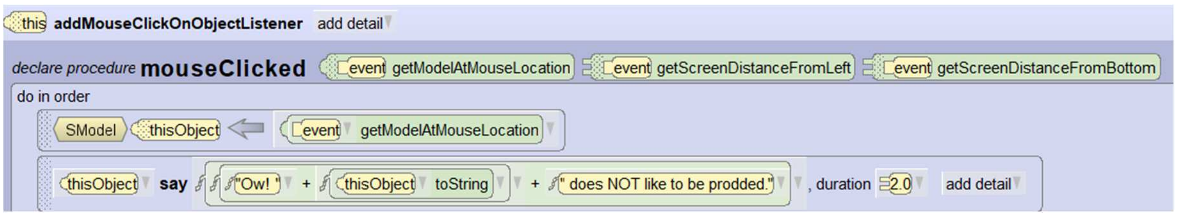
```
this addMouseClickOnObjectListener add detail
declare procedure mouseClicked  event getModelAtMouseLocation  event getScreenDistanceFromLeft  event getScreenDistanceFromBottom
do in order
    drop statement here
```

The parameters are:

- event.getMouseModelLocation
- event.getScreenDistanceFromLeft
- event.getScreenDistanceFromBottom

getModelAtMouseLocation gets the object that was clicked. Since this could be any object, Alice doesn't know what class it will belong to. If we wanted to create a variable to hold this so that we can easily access its methods and functions, the variable should be created as gallery class SModel, which is the superclass of Biped, Quadruped etc, and has most of their procedures and functions.

An example of what the code could look like in this method is:



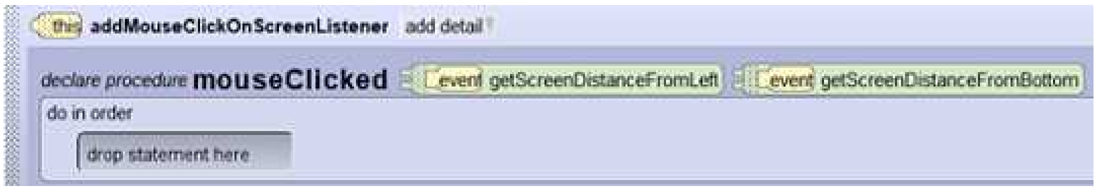
```
this addMouseClickOnObjectListener add detail
declare procedure mouseClicked  event getModelAtMouseLocation  event getScreenDistanceFromLeft  event getScreenDistanceFromBottom
do in order
    SModel thisObject ⟸ event getModelAtMouseLocation
    thisObject say "Ow! " + thisObject toString + " does NOT like to be prodded." , duration 2.0 add detail
```

Notes:

- We would create the variable thisObject as data type SModel
- Once created, we can drag in the parameter **getModelMouseLocation** as its initial value
- **toString** is a function of **SModel**, and it returns the object's name. We can use it as seen here, and we can also use it to determine which object was clicked.

5.3.3 AddMouseClickOnScreenListener

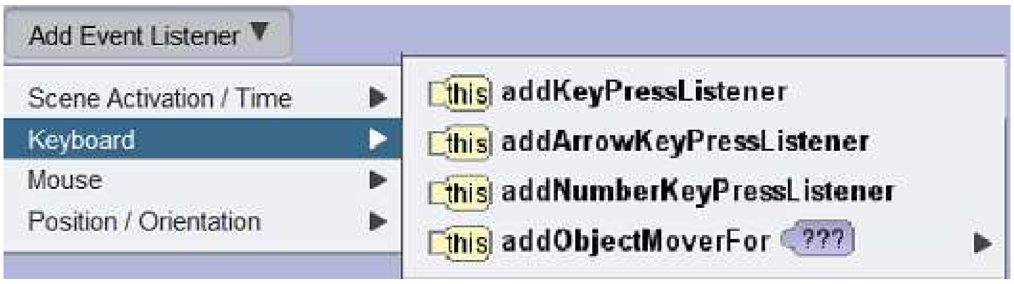
This is triggered if the user clicks anywhere on the screen. The method created looks like this:


this addMouseClickOnScreenListener add detail
declare procedure mouseClicked event getScreenDistanceFromLeft event getScreenDistanceFromBottom
do in order
drop statement here

You could use the two parameters to find out which part of the screen the user clicked on.

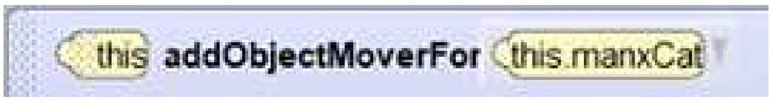
5.4 Keyboard Events

Again, there are a few different types of keyboard event listeners.



The simplest is **addObjectMoverFor**. It takes a single parameter: the object that can be moved by the user, and does not allow extra code to be added to it.

The method will look like this:

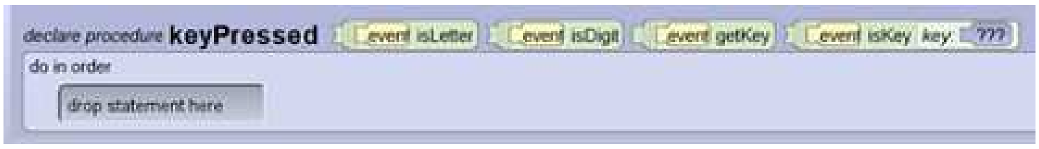

this addObjectMoverFor this.manxCat

This allows the user to move this object left and right with the arrow keys, and forward and back with the up and down arrow keys. It does not allow the object to be moved up and down or turned.

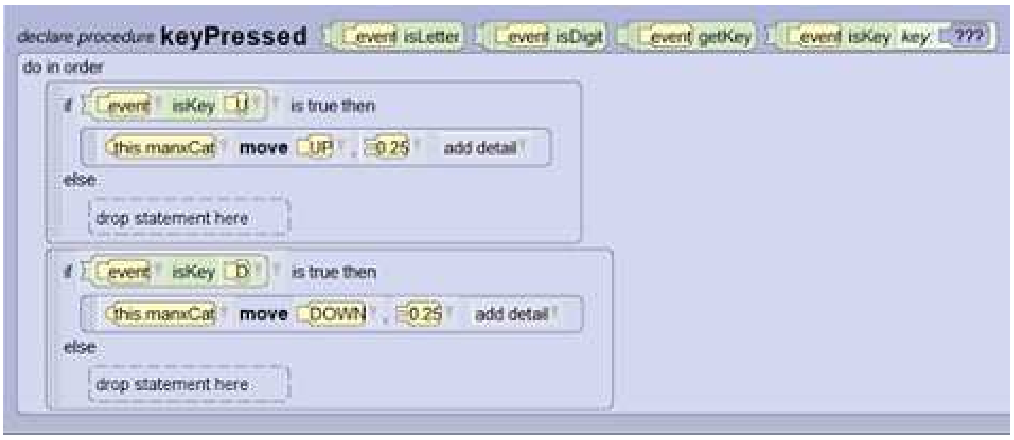
The other three listeners are very similar to each other, so we will only look at one of them. **addKeyPressListener** listens for any key from the keyboard; **addArrowKeyPressListener** only listens for use of the arrow keys, and **addNumberKeyPressListener** only listens for the numbers 0 to 9.

Let's look at **addKeyPressListener**.

When created, the method looks like this:


declare procedure keyPressed event isLetter event isDigit event getKey event isKey key: ???
do in order
drop statement here

You can use the four parameters **isLetter**, **isDigit**, **getKey** and **isKey** to get more information about the key that was pressed. The example below adds up and down movement to the Manx cat, in addition to the movements allowed by the **addObjectMoverFor** method.



```
declare procedure keyPressed  event isLetter  event isDigit  event getKey  event isKey key: ???
do in order
  if  event  isKey U  is true then
    this.manxCat  move  UP , 0.25  add detail
  else
    drop statement here
  if  event  isKey D  is true then
    this.manxCat  move  DOWN , 0.25  add detail
  else
    drop statement here
```

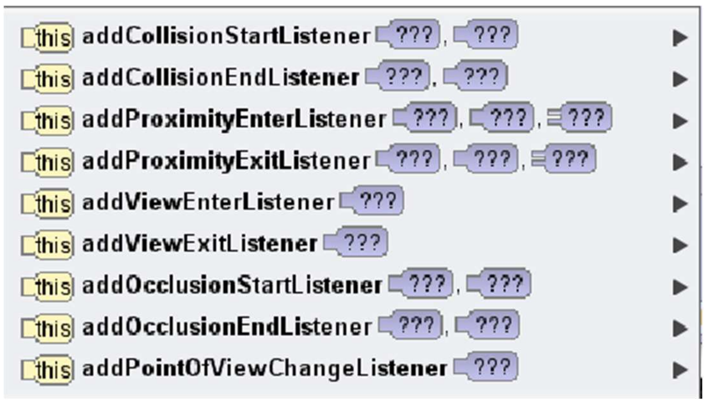
Note that the **isKey** parameter has been dragged down into each of the two conditions in the method.

5.5 Position/Orientation Events

These events happen when an object changes position to a place where a condition it satisfied, for example, it has collided with another object. These listeners can be used for many things in a game, for example:

- See if the hero has been hit by a spear
- See if a UFO has collided with an alien robot
- See whether an aeroplane has reached the ground
- See if a ball has entered a goal

Selecting 'Position/Orientation' event gives the following choices:

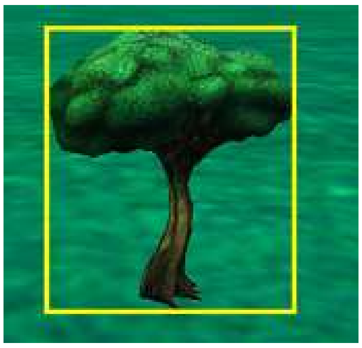


```
this addCollisionStartListener ???, ???
this addCollisionEndListener ???, ???
this addProximityEnterListener ???, ???, ???
this addProximityExitListener ???, ???, ???
this addViewEnterListener ???
this addViewExitListener ???
this addOcclusionStartListener ???, ???
this addOcclusionEndListener ???, ???
this addPointOfViewChangeListener ???
```

First, let's look at what the terms actually mean:

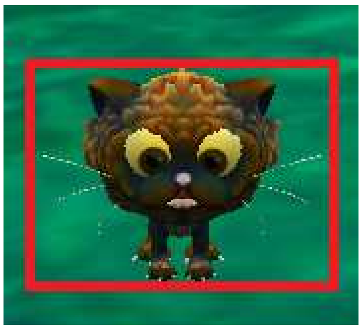
Collision:	Two objects touch each other
Proximity:	Two objects are within a given distance of each other
View:	An object is able to be seen by another. For this to be true, the viewing object must be facing the right way, and there must be no other object blocking the view.
Occlusion:	An object is not within view
Point of View Change:	An object has either moved or turned, so its view is different

Since all of these work in a similar way, we will only look at collisions. The way collisions behave can be confusing unless you understand the concept of bounding boxes. Each object has a bounding box, which defines the space it actually occupies. In the image below, the bounding box of the tree (in two dimensions) may be as shown by the yellow rectangle.

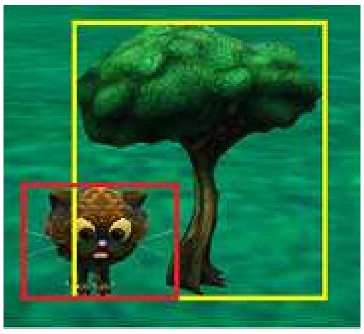


The object actually occupies all the space within the bounding box, and a collision occurs when the bounding boxes of the objects touch each other.

A cat's bounding box may look like this – remember, the whiskers are included.

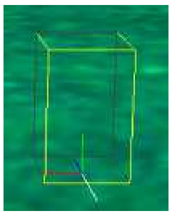


If the cat and the tree are positioned as shown in the image below, they are colliding, because their bounding boxes are touching.



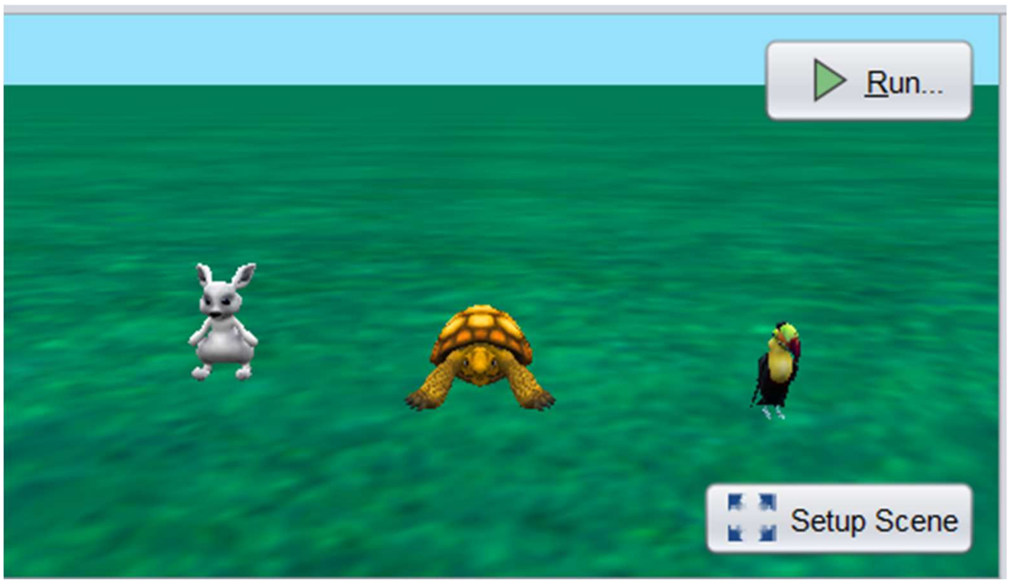
A trick: if you wanted the collision only to occur if the cat touched the trunk of the tree, you could create a small object (maybe one of the shapes), position it at the same place as the tree trunk, set its opacity to 0, and monitor the collision between the shape and the cat, rather than the tree and the cat.

In fact, as you're dragging a new object into a scene, you actually see its bounding box as you're positioning it:

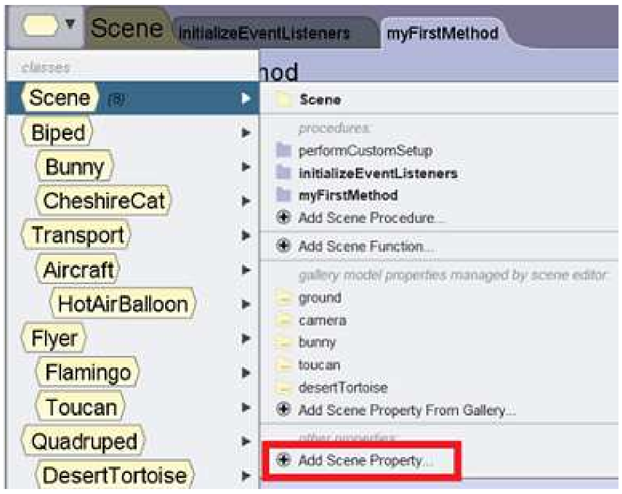


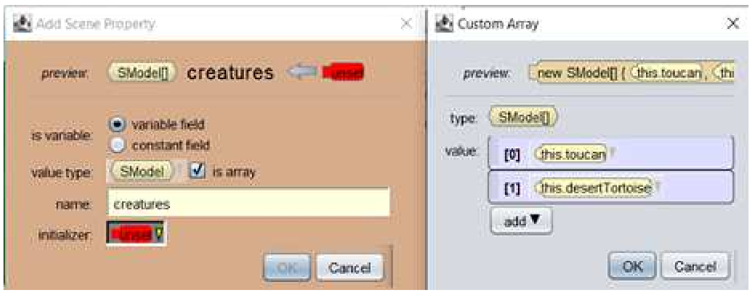
Let’s now look at adding a collision start listener.

The scene in this example looks like this, with three objects: a bunny, a tortoise and a toucan. The user will be able to move the bunny. If the bunny collides with any of the others, the bunny will say ‘Ow!’ and the other object will say, ‘Mind where you’re going!’



First we’ll create an array containing the tortoise and the toucan. We’ll use this when we create the collision listener. We’ll create it as a property of the scene.

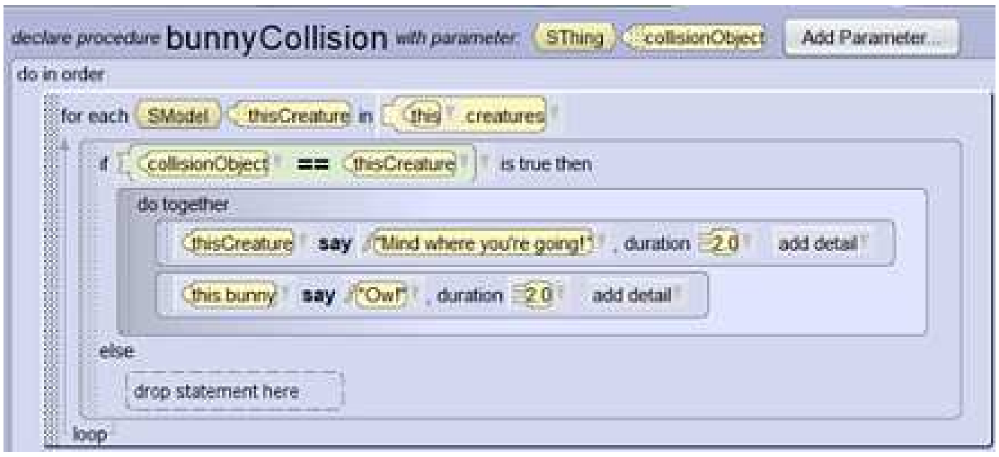



Add Scene Property
preview: SModel[] creatures ⇐ unset
is variable: ● variable field ○ constant field
value type: SModel ☑ is array
name: creatures
initializer: unset
OK Cancel

Custom Array
preview: new SModel[] { this.toucan, thi
type: SModel[]
value: [0] this.toucan
[1] this.desertTortoise
add ▼
OK Cancel

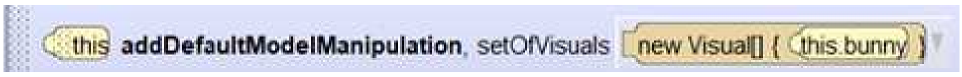
We can use **SModel** as the value type, because it is the superclass of all bipeds, quadrupeds and flyers.

Next we'll code a procedure that will be called when there is a collision. The procedure will be created under the scene. The procedure looks like this:

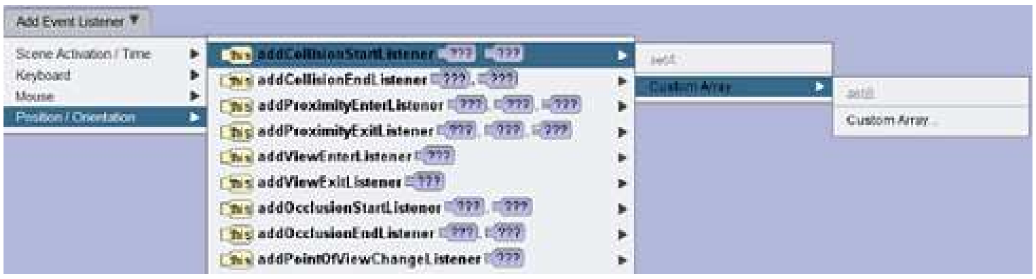

```
declare procedure bunnyCollision with parameter: SThing collisionObject   Add Parameter...
do in order
  for each SModel thisCreature in this creatures
    if collisionObject == thisCreature is true then
      do together
        thisCreature say "Mind where you're going!" , duration 2.0 add detail
        this.bunny say "Ow!" , duration 2.0 add detail
    else
      drop statement here
  loop
```

It has one parameter, which will be the object that the bunny collided with. As we'll see when we code the collision event listener, it has a parameter holding the object that has been collided with, and we'll pass that parameter to this procedure. However, Alice defines this parameter as class **SThing**, which is a superclass of all objects. **SThing** has no 'say' method, so in this procedure we need to find the actual object of class **SModel**, and use its 'say' method. We therefore compare the parameter **collisionObject** with all the objects in **creatures** array to find the correct object of class **SModel.**

In the InitializeEventListeners tab, we'll add default model manipulation for the bunny, so he can be moved.

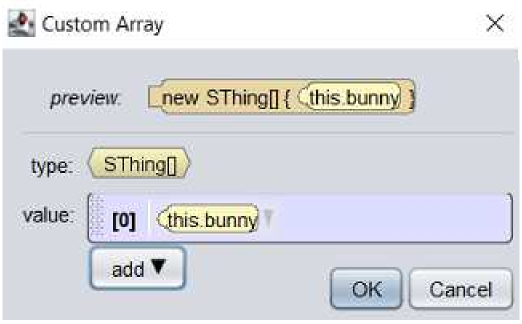

```
this addDefaultModelManipulation, setOfVisuals new Visual[] { this.bunny }
```

Next we want to add a Collision Start Listener to listen for a collision between the bunny and either the tortoise or the toucan.


Add Event Listener
Scene Activation / Time
Keyboard
Mouse
Position / Orientation
this addCollisionStartListener ??? ???
this addCollisionEndListener ??? ???
this addProximityEnterListener ??? ??? ???
this addProximityExitListener ??? ??? ???
this addViewEnterListener ???
this addViewExitListener ???
this addOcclusionStartListener ??? ???
this addOcclusionEndListener ??? ???
this addPointOfViewChangeListener ???
set
Custom Array
Custom Array...

The collision listeners allow us to monitor collisions between any objects in one array known as Set A and any objects in another array known as Set B.

We'll choose a custom array for set A, and the **creatures** array for set B. For Set A, it will then pop up a box that allows us to add items to the array. We will add only the bunny.



Custom Array

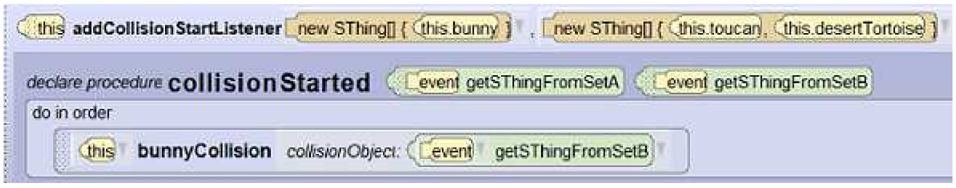
preview: new SThing[] { this.bunny }

type: SThing[]

value: [0] this.bunny

add ▼ OK Cancel

In this listener, we'll call the procedure **bunnyCollision**, passing as a parameter the object from Set B that the bunny has collided with. We can use the **getSThingFromSetB** parameter to get this object.



```
this addCollisionStartListener new SThing[] { this.bunny } , new SThing[] { this.toucan , this.desertTortoise }
  declare procedure collisionStarted  ( event getSThingFromSetA )  ( event getSThingFromSetB )
  do in order
    this bunnyCollision collisionObject: event getSThingFromSetB
```

6 Finding more information

This course was designed to get you started with Alice programming, but if you enjoyed it and want to learn more, the Alice official website has lots of useful information and how-to guides.

Visit http://www.alice.org/resources/ and explore! The how-to guides have details on how to use all the Alice features. You'll also find lots of programs that you can download, so you can see what other people have done and get ideas for your own programs.

There's a community of Alice enthusiasts who are happy to share ideas and answer questions here: http://www.alice.org/forums/

And if you search on Youtube for 'Alice programming', you'll find a lot of helpful videos.

Enjoy!