Learning Java through Alice 3
3rd Edition

```
public static void main(String[] args) {
  Model[] characters = new Model[]{madHatter, whiteRabbit,
    cheshireCat, queenOfHearts, marchHare};
  for (int i = 0; i < characters.length; i++) {
    characters[i].say("My name is " + characters[i].getName());
  }
}
```

- An Introduction to Programming

Tebring Daly and Eileen Wrigley

ISBN-13: 978-1514278901

Table of Contents

[*] *Ongoing project*

* *Ongoing project*

* *Ongoing project*

* *Ongoing project*

* *Ongoing project*

Acknowledgments

First and foremost, the authors would like to convey special thanks to the Alice Software team at Carnegie Mellon University for providing the Alice software to make this text possible. Also, we would like to take this opportunity to thank Electronic Arts, Inc. for providing their rich set of graphics which certainly makes Java and Alice programming more interesting. In addition, we would like to express our gratitude to the National Science Foundation and the Alice ATE grant team members for their continued support. Words cannot express our gratitude to Wanda Dann who was abundantly helpful and offered invaluable assistance, encouragement, and guidance.

A heartfelt thanks to those colleagues that helped with revisions to this book: Bob Benavides (Computer Science Professor at Collin College in Plano, TX), Branden Simbeck (Computer Information Technology Professor at Community College of Allegheny County), and Rod Farkas (Computer Information Technology Professor at Community College of Allegheny County).

About the Authors

This book has been a joint effort by a mother and daughter team.

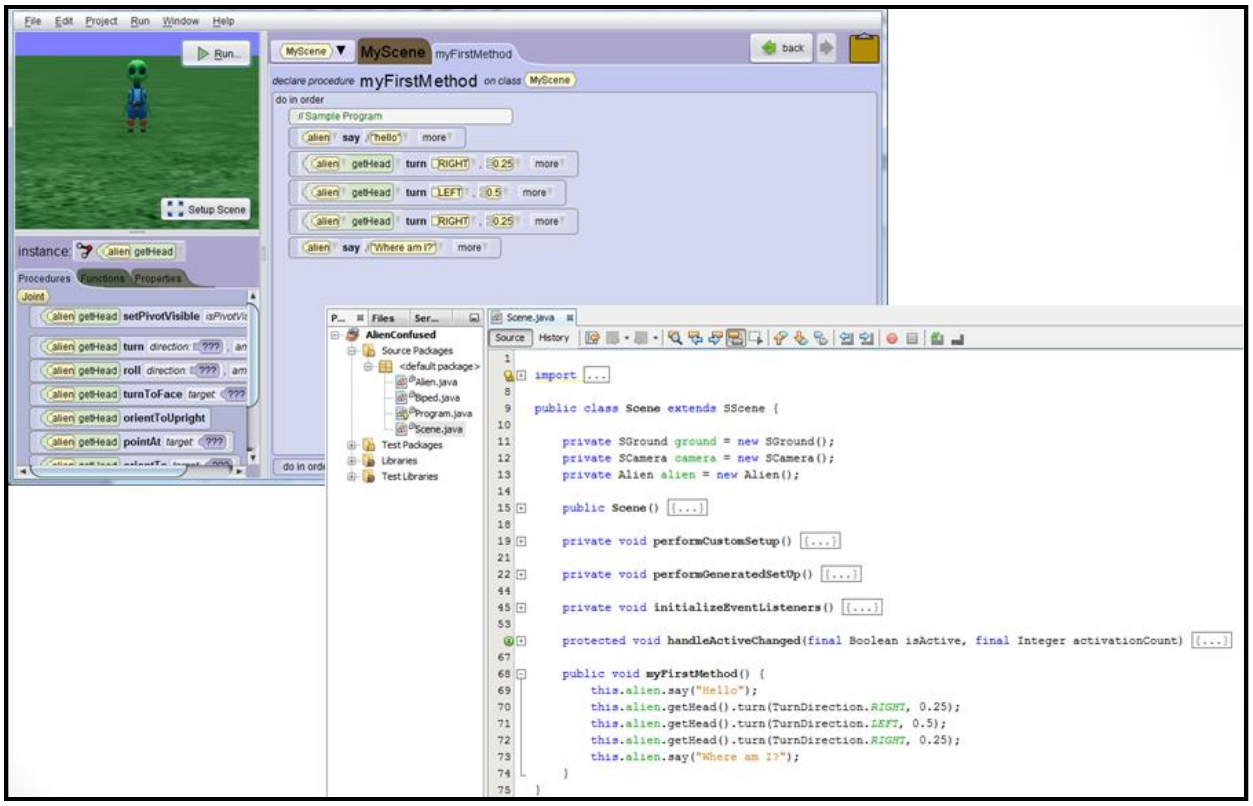
Eileen Wrigley, full-time Professor of Computer Information Technology courses at the Community College of Allegheny County in Pittsburgh, Pennsylvania brings more than 40 years of teaching experience to her writing. She earned her B.S. and M.S. degrees from the University of Pittsburgh in Mathematics and Computer Science.

Dr. Tebring Daly has been teaching full-time in the Computer Science department at Collin College in Plano, Texas since 2006. She has earned her B.S. and M.S. degrees from the University of Pittsburgh and Ph.D. from the University of North Texas. Professor Daly has been teaching Java courses for 9 years.

Approach

This book is designed for students wanting to learn fundamental programming concepts. No previous programming experience is required. All of the software used in this text are available to download free of charge. The versions of the software may differ slightly from the versions used in this text since the versions are constantly being updated.

This book will teach you how to program by using Java code. We will use a Java editing tool called NetBeans[1] to help write the code. The environment can be used on a Windows® operating system[2], Apple Macintosh® operating system[3] (Mac), or Linux® operating system[4].
We are also using a tool called Alice 3[5], to provide visuals for abstract programming concepts. This environment works on Windows, Mac, and Linux operating systems. Alice 3 is a drag and drop environment that can be transferred into Java code in the NetBeans environment as shown below.


File Edit Project Run Window Help
Run...
Setup Scene
MyScene
MyScene myFirstMethod
back
declare procedure myFirstMethod on class MyScene
do in order
// Sample Program
alien say "hello" more
alien getHead turn RIGHT , 0.25 more
alien getHead turn LEFT , 0.5 more
alien getHead turn RIGHT , 0.25 more
alien say "Where am I?" more
instance: alien getHead
Procedures Functions Properties
Joint
alien getHead setPivotVisible isPivotVis
alien getHead turn direction: ??? , am
alien getHead roll direction: ??? , am
alien getHead turnToFace target: ???
alien getHead orientToUpright
alien getHead pointAt target: ???
AlienConfused
Source Packages
<default package>
Alien.java
Biped.java
Program.java
Scene.java
Test Packages
Libraries
Test Libraries
Scene.java
Source History

```
import ...

public class Scene extends SScene {

    private SGround ground = new SGround();
    private SCamera camera = new SCamera();
    private Alien alien = new Alien();

    public Scene() {...}

    private void performCustomSetup() {...}

    private void performGeneratedSetUp() {...}

    private void initializeEventListeners() {...}

    protected void handleActiveChanged(final Boolean isActive, final Integer activationCount) {...}

    public void myFirstMethod() {
        this.alien.say("Hello");
        this.alien.getHead().turn(TurnDirection.RIGHT, 0.25);
        this.alien.getHead().turn(TurnDirection.LEFT, 0.5);
        this.alien.getHead().turn(TurnDirection.RIGHT, 0.25);
        this.alien.say("Where am I?");
    }
}
```

[1] Supported by Oracle, http://netbeans.org
[2] Microsoft Corporation, http://www.microsoft.com
[3] Apple Macintosh Corporation, http://www.apple.com/osx
[4] Linux Foundation, http://www.linux.org
[5] Developed by Carnegie Mellon University (CMU), Alice, http://www.alice.org

Chapter Breakdown

Preface provides an overview of the text.

Chapter 0 will help you download the required software and introduce you to the Alice 3 environment.

Chapter 1 describes the history of Java, basic programming terminology, and provides hands-on practice coding in the Alice 3 environment, writing Java code from scratch in NetBeans, and transferring Alice 3 projects into the NetBeans environment to look at the Java code.

Chapter 2 covers naming rules, creating and using variables in code, using arithmetic statements, order of operation, shorthand operators, and casting rules.

Chapter 3 explains various ways of formatting output and receiving user inputs. The user is introduced to import statements.

Chapter 4 shows the user how to modularize programs using procedural methods. The user is introduced to the Java documentation and the syntax for writing procedural methods.

Chapter 5 expands upon chapter 4 to include methods that return values which are known as functional methods.

Chapter 6 provides an explanation and practice with relational and logical operators using conditionals.

Chapter 7 talks about object-oriented terms (encapsulation, inheritance, and polymorphism) and their use.

Chapter 8 provides an introduction to GUI and the structure for creating basic drawings.

Chapter 9 shows three types of repetition techniques (while, do while, and for loop).

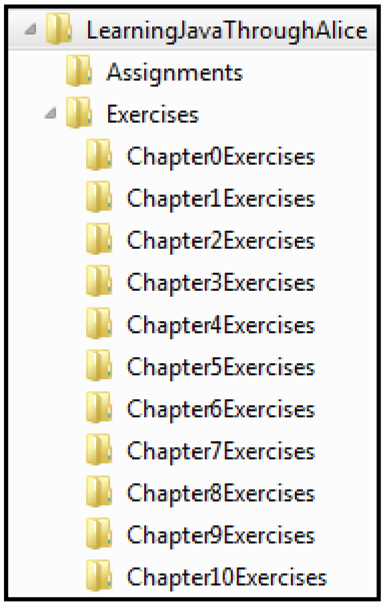
Chapter 10 explains how to use an array to store multiple values of the same type.

Organization

Each chapter is divided into content segments, hands-on exercises, a summary, and review questions. You should work through the hands-on exercises in each chapter.

The data files needed for the hands-on exercises and assignments can be found at http://iws.collin.edu/tdaly/book3.

You may want to create an organizational method for keeping track of your files. Each chapter has several exercises that will walk you through the programming concepts for that chapter. There is at least one assignment at the end of every chapter. The assignments test your ability to put the concepts from the chapter into action on your own. Please save the chapter exercises to the "Exercises" folder and the assignments at the end of each chapter to the "Assignments" folder so that you don't get confused.



Instructors: Please email tdaly@collin.edu for solutions, sample syllabi, etc.

Chapter 0

Getting Started

Objectives

- ☑ Install the Java editor
- ☑ Install the Alice environment
- ☑ Setup the Java editor to work with the Alice environment
- ☑ Explain the purpose of Alice
- ☑ Setup an Alice scene

Installing the Java and NetBeans

Java is an object-oriented programming language. We will be writing all of our Java code in NetBeans. NetBeans is not the only environment for writing Java code, but it is what we will be using for this text.

You should download the NetBeans and Java SDK (Software Development Kit) bundle. This bundle will include everything that you will need to write and run Java programs. Please follow the install directions located on the following website: http://iws.collin.edu/tdaly/book3

Installing the Alice Environment

Alice 3 provides a 3D environment for manipulating objects using drag and drop code segments. This environment helps to provide visual representations of abstract programming concepts. Please follow the install directions located on the following website: http://iws.collin.edu/tdaly/book3

Setting up NetBeans to Work with Alice

There is an Alice 3 plugin file that you will also need to download and add to the NetBeans environment. Please follow the install directions located at the following website: http://iws.collin.edu/tdaly/book3

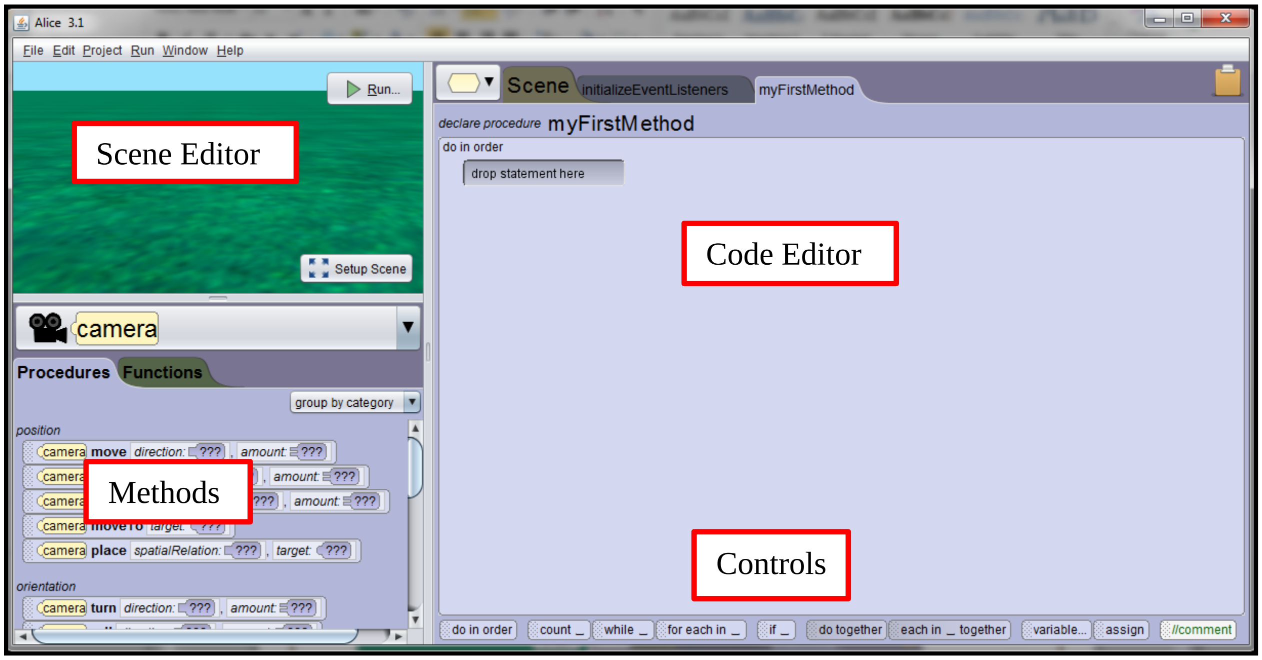
What is Alice?

The Alice team at Carnegie Mellon University named the Alice programming software in honor of Lewis Carroll who wrote Alice's Adventures in Wonderland. Lewis Carroll was able to do complex mathematics and logic, but he knew that the most important thing was to make things simple and fascinating to a learner.

Alice makes it easy to create an animation or interactive game. It is designed for beginners who want to learn object-oriented programming. In Alice, 3-D objects (e.g., people, aliens, animals, props) are placed in a scene. Then, students drag and drop tiles to create a program to animate the objects. These tiles correspond closely to statements in Java. Alice allows student to immediately see how their programs run, enabling them to easily understand the relationship between the programming statements and the behavior of objects in their program.

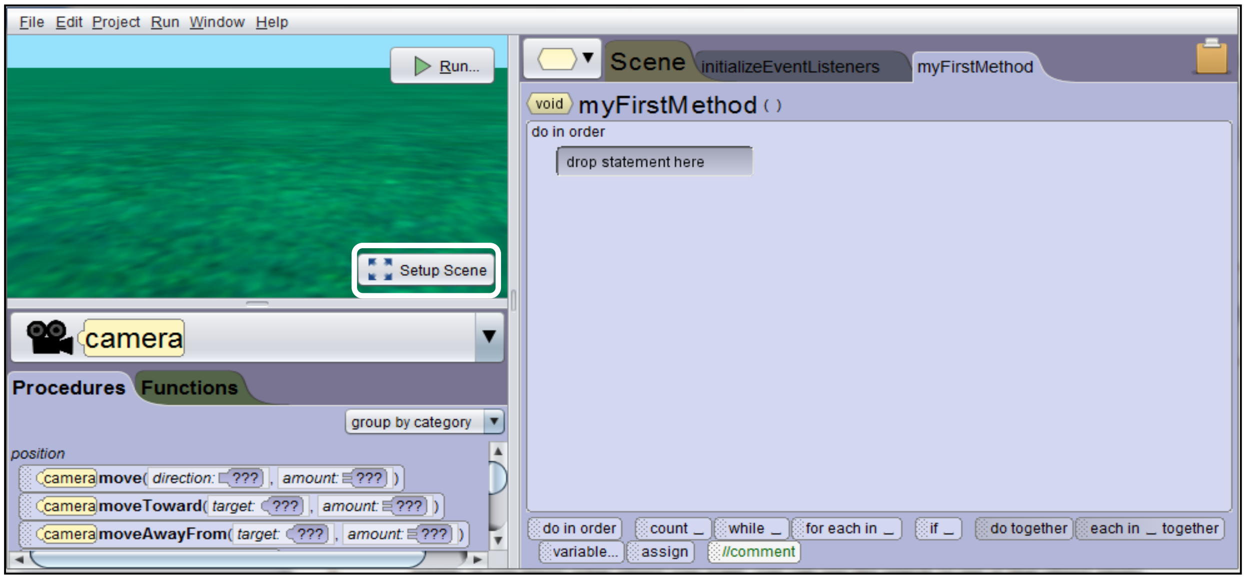
Alice 3 is the newest version of the Alice software. This version of the software allows users to transfer Alice projects into the NetBeans environment to edit the Java code. This text will be using Alice to demonstrate fundamental programming concepts in Java such as objects, methods, looping, etc. by creating animations.

An Alice scene begins with a template for an initial scene. These templates can be grass, water, snow, etc. Then, you add various objects to the scene to create the virtual scene that you desire.

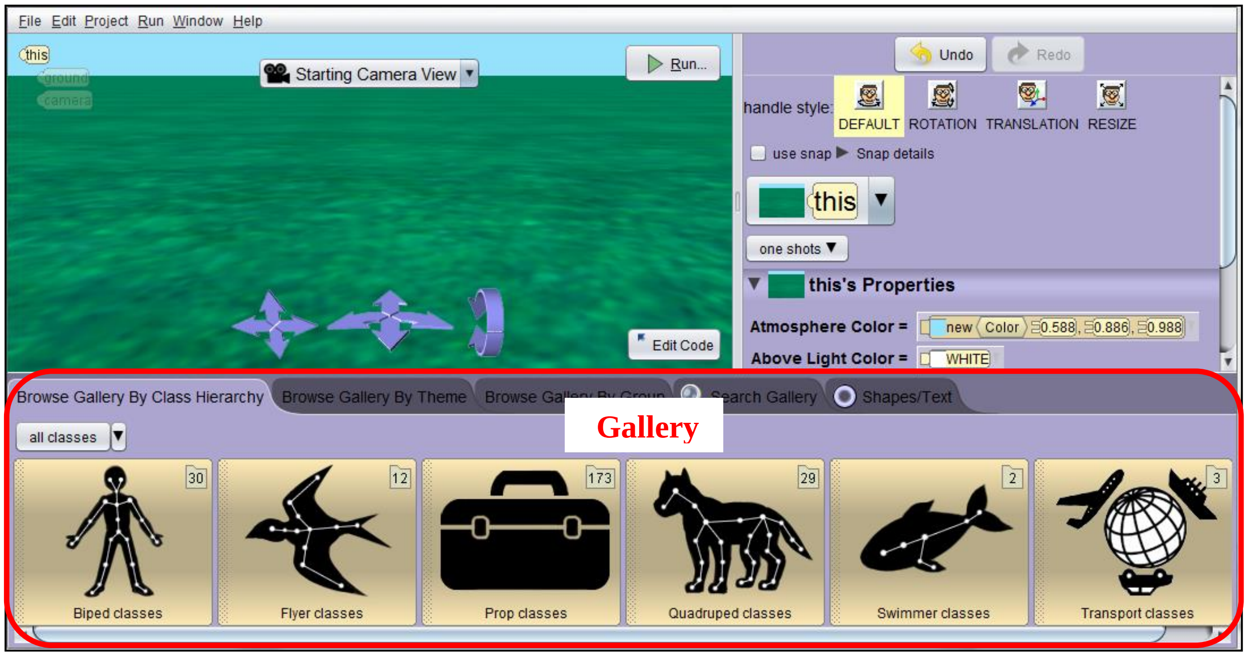

Alice 3.1
File Edit Project Run Window Help
Run...
Scene Editor
Setup Scene
camera
Procedures Functions
group by category
position
camera move direction: ??? , amount: ???
camera ... , amount: ???
camera ... ??? , amount: ???
camera moveTo target: ???
camera place spatialRelation: ??? , target: ???
orientation
camera turn direction: ??? , amount: ???
Methods
Scene initializeEventListeners myFirstMethod
declare procedure myFirstMethod
do in order
drop statement here
Code Editor
Controls
do in order
count _
while _
for each in _
if _
do together
each in _ together
variable...
assign
//comment

Alice Scene Setup

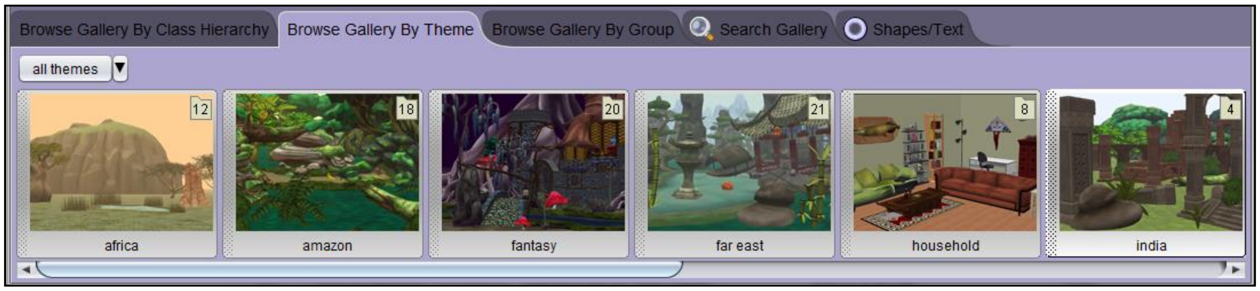
Objects are added to the scene via the scene editor (Click on Setup Scene button).

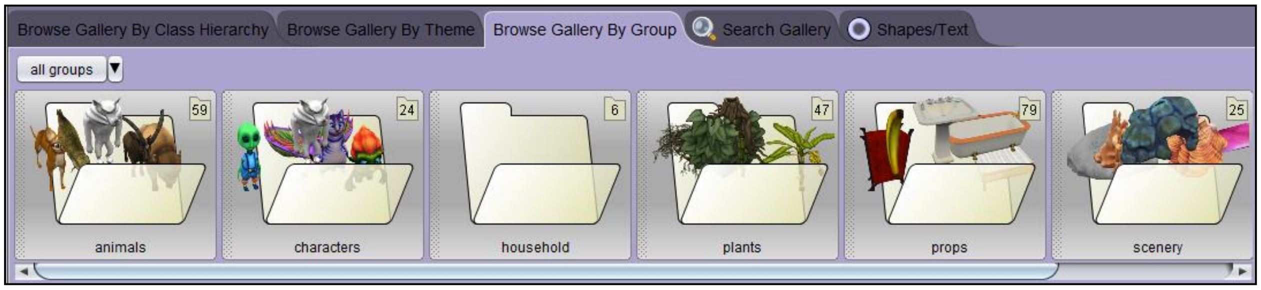


There are several choices for selecting objects from the gallery. The **hierarchy** choice is broken down by physical makeup. A biped has 2 legs, a flyer has wings, a prop is something that is inanimate, a quadruped has 4 legs, and a swimmer has fins.

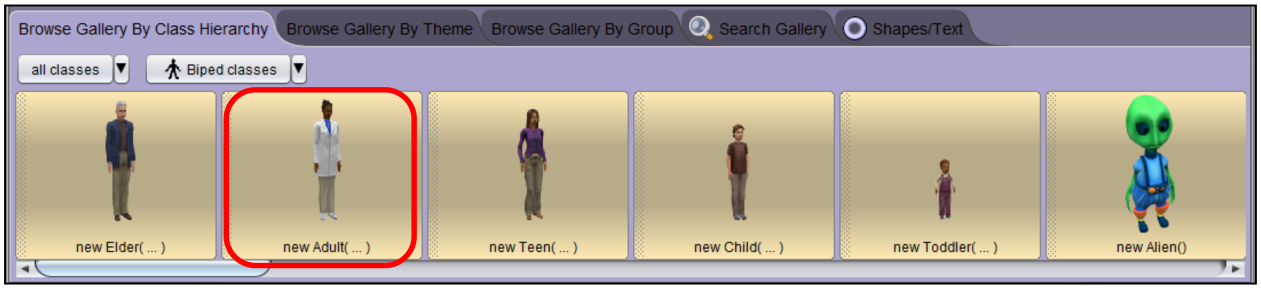


If you want you can also view the objects by **theme** or by **group** as shown below. The **search** feature is nice if you are looking for a particular object. If you want to add a 2D image to your world you could add a billboard object (located in the **shapes/text** tab) and change the image to an image that you have saved (this works nicely for background images).

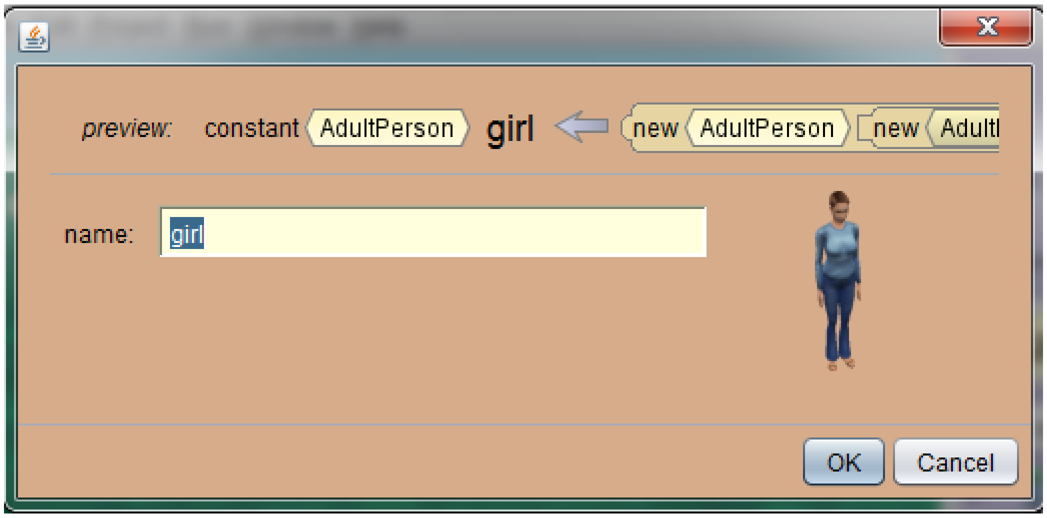




The Alice developers have provided a number of 3D models for you to use in your animations. **An Alice 3D model (class) is a blueprint that tells Alice how to create a new object in the scene.** The 3D model provides instructions on how to draw the object, what color it should be, what parts it should have, its size (height, width, and depth), and many other details. Once you decide what objects, you would like to have, you will need to click on the class to create an object of that type. For example, if I want a girl object in my world, I would select the Biped folder and then the Adult class to create the girl object.

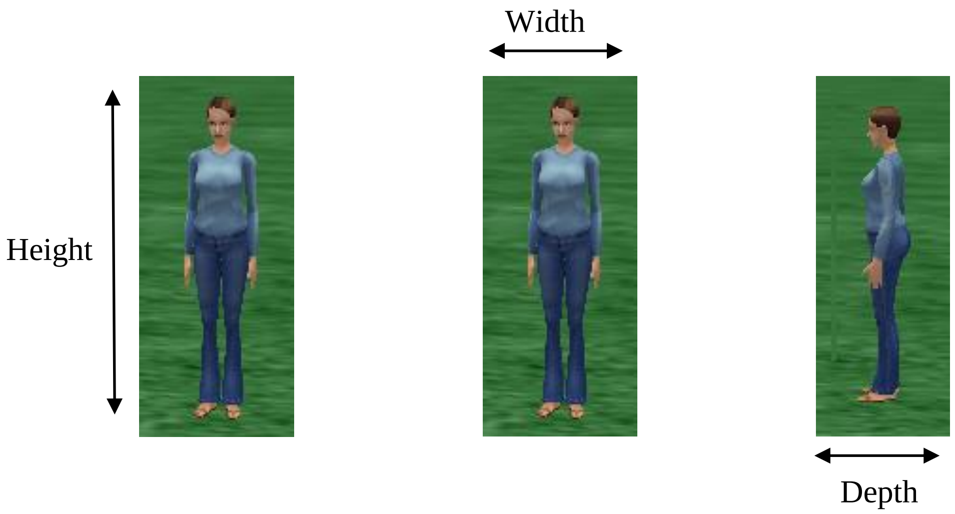


When you create an object, you will need to give it a name. You can leave the default name or give it your own name. You cannot give two objects the same name. Be careful when you are creating objects, if you try and use the name girl more than once, it won't let you create the new object.


preview: constant AdultPerson girl ⇐ new AdultPerson new Adult
name: girl
OK
Cancel

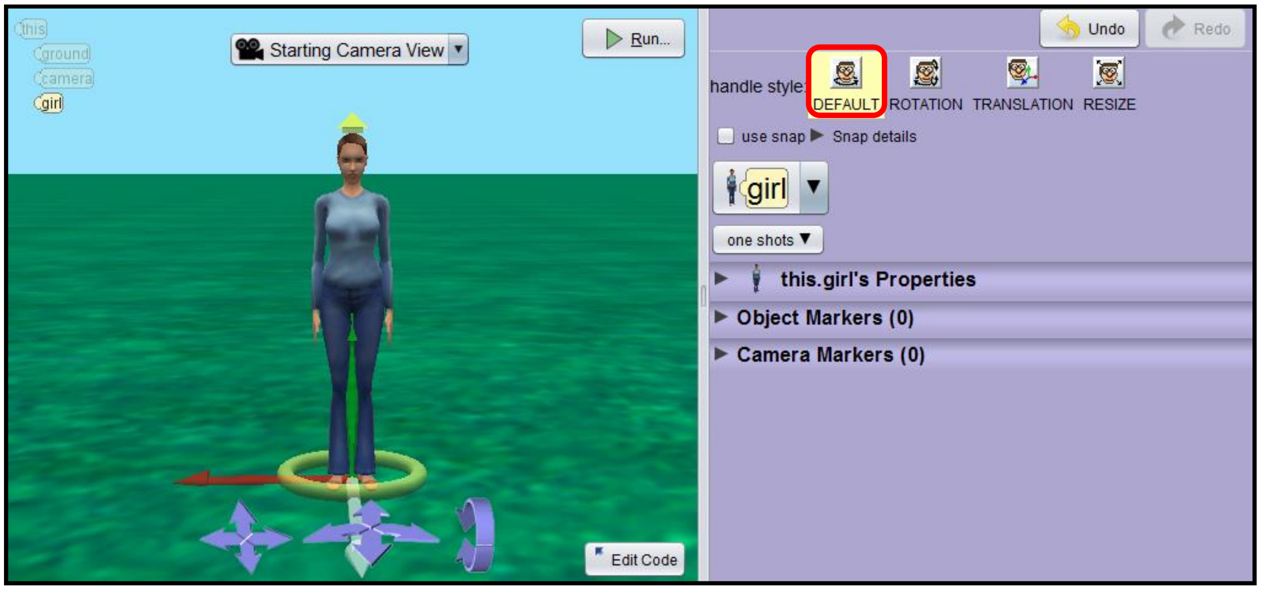
All objects will initially be placed into the middle of the scene and then can be manipulated to any position desired. Alternatively, objects can be dragged to any position in the scene.

Alice objects are represented in a 3 dimensional space. Each object has width, height, and depth as shown below. The height is measured vertically, the width is measured horizontally, and the depth is measured from front to back.

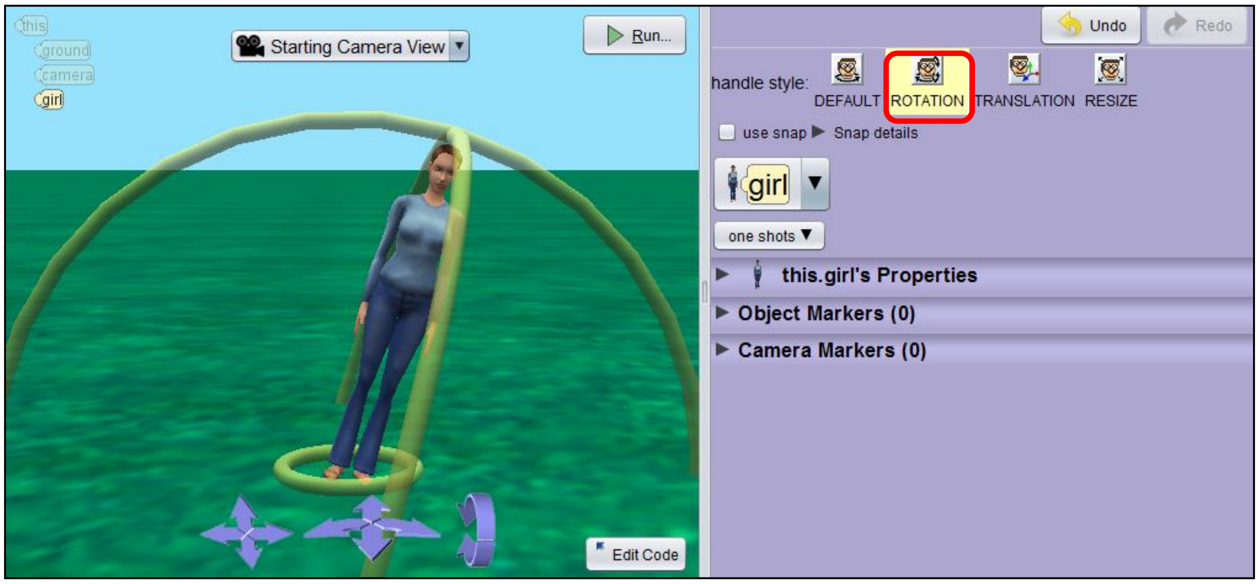


There are six possible directions in which an object may move – **forward, backward, up, down, left and right**. Remember that directions are left and right with respect to the object, not the camera's point of view. For example, this girl object can move forward, backward, up (in air), down (into ground), her left, or her right. The direction an object is facing and where the top of the object is located (relative to the world) is known as the **object's orientation**. In the scene editor, there are 4 buttons that allow you to manipulate the object.

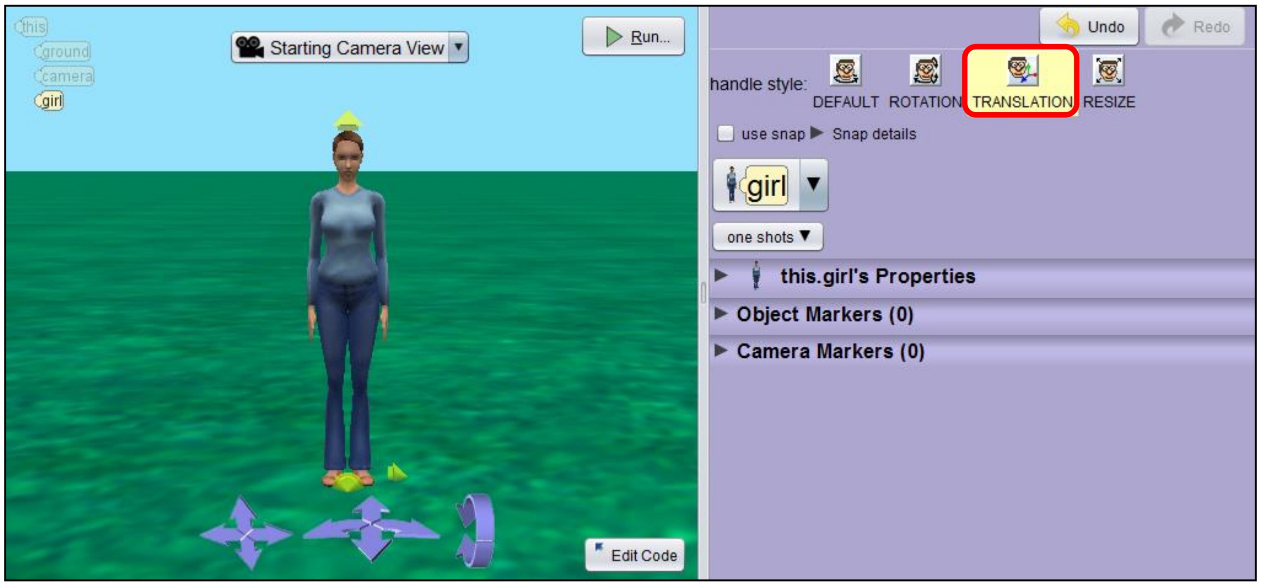
The **DEFAULT** button allows you to move the object and do some rotations. Hold down the left mouse button and drag the circle to rotate the object. To move the object, hold down the left mouse button and drag the object wherever you want.



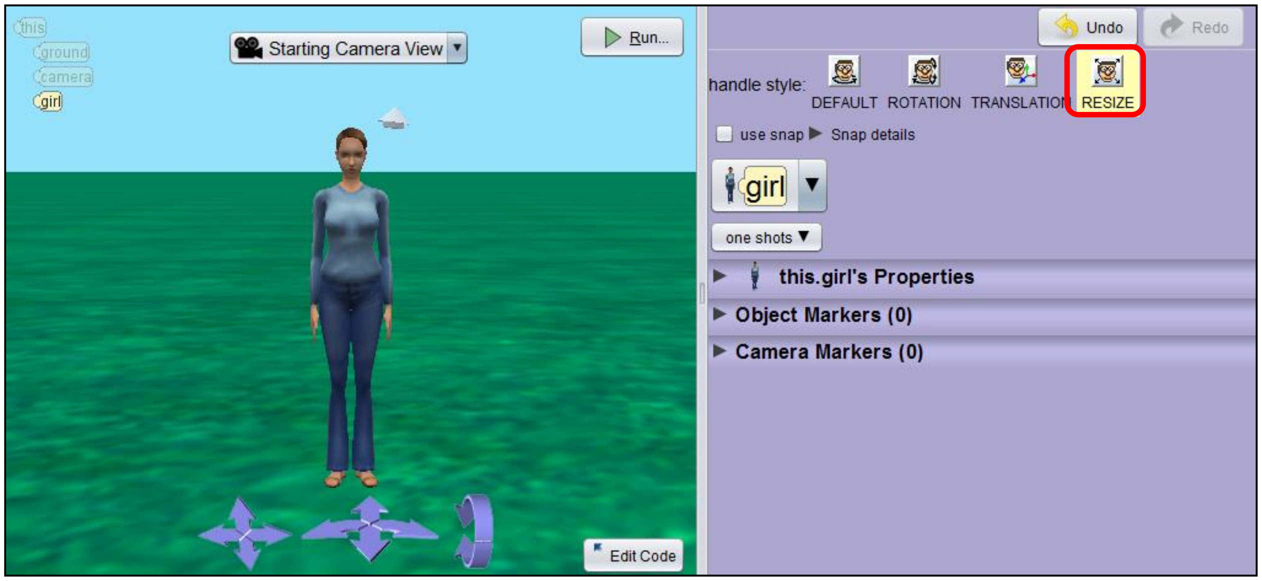
The **ROTATION** button allows you to do rotations in all directions. Hold down the left mouse button and drag the appropriate circle to rotate the object.



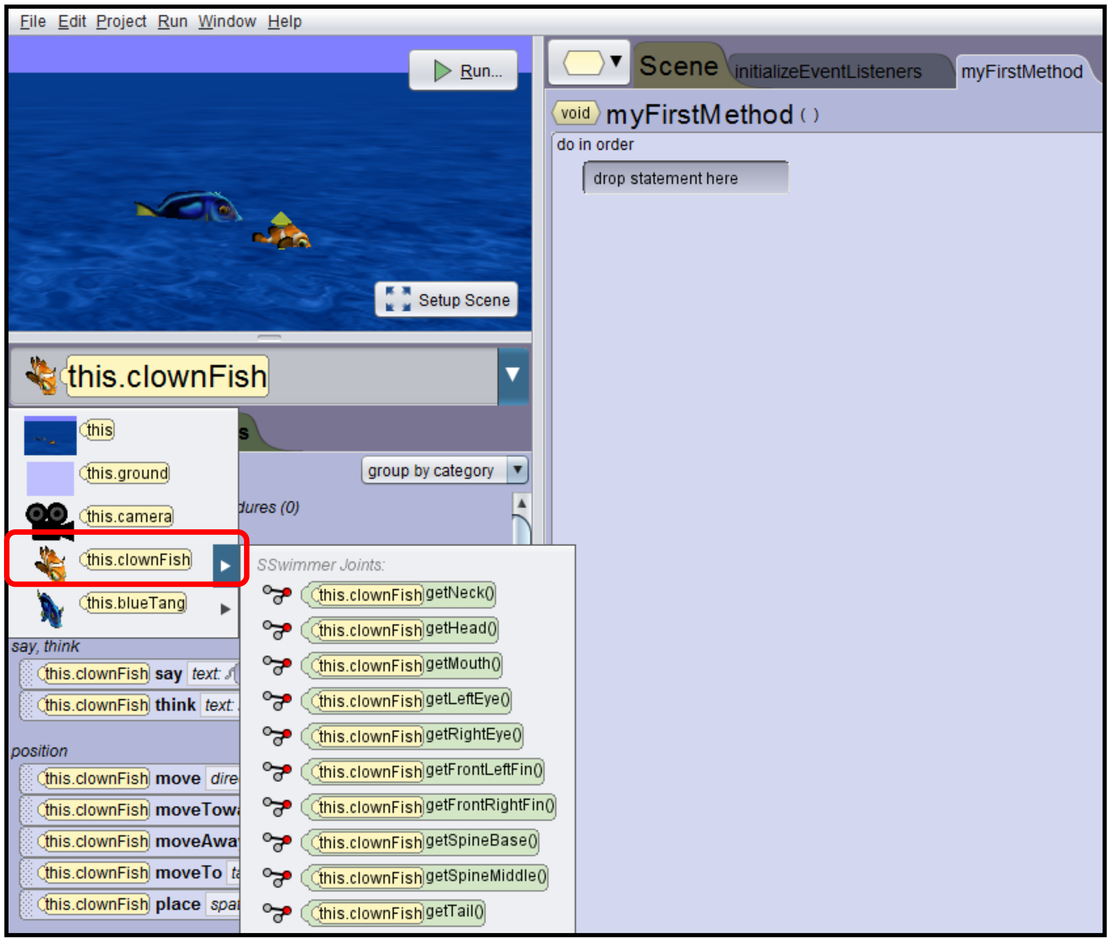
The **TRANSLATION** (move) button allows you to move the object in all directions. Hold down the left mouse button and drag the arrows to move the object. The arrow at the top of the object moves the object up and down, the arrow on the right of the object moves the object left and right, and the arrow in front of the object moves the object forward and backward.



The **RESIZE** button allows you to resize the object. Hold down the left mouse button and drag the arrow at the top of the object. The object will resize proportionately.



All of the Alice models have body parts that can be manipulated with rolls, turns, etc. You can access the subparts for an object by clicking the part drop down next to the object drop down.



The best way to see how Alice 3 works is to create a virtual world with objects and animate the objects in that world. This will be done in the hands-on exercises.

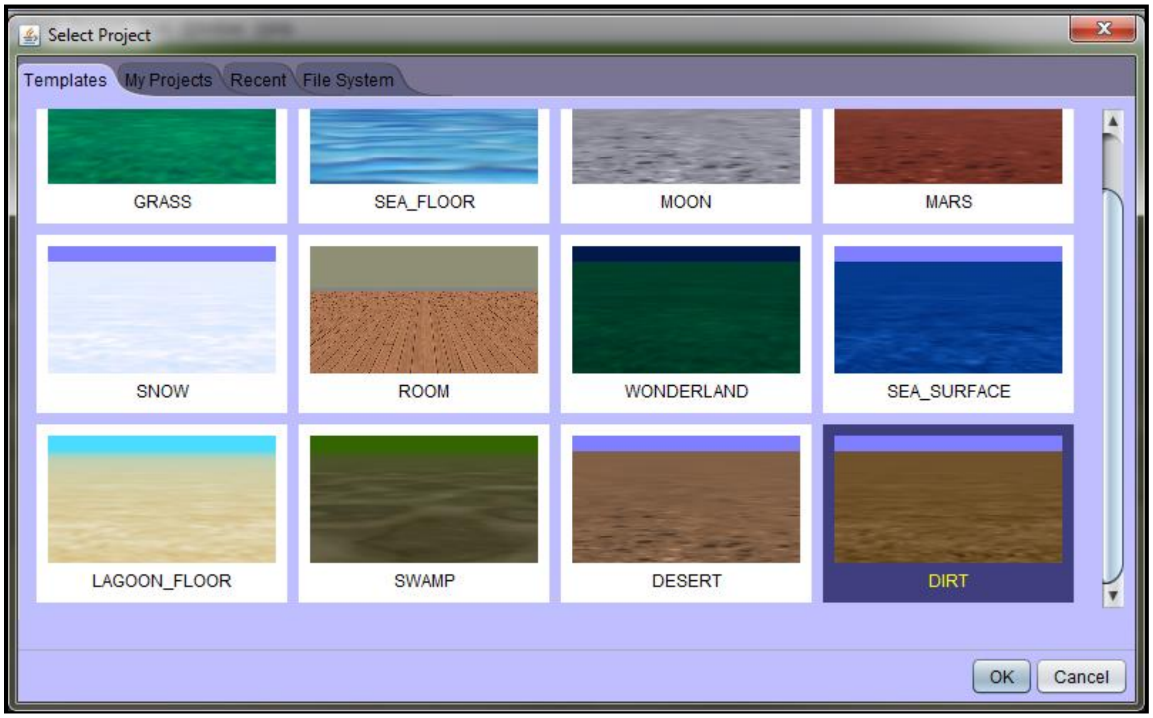
Hands-on Exercises

Exercise 1: Manipulating the Alice Environment

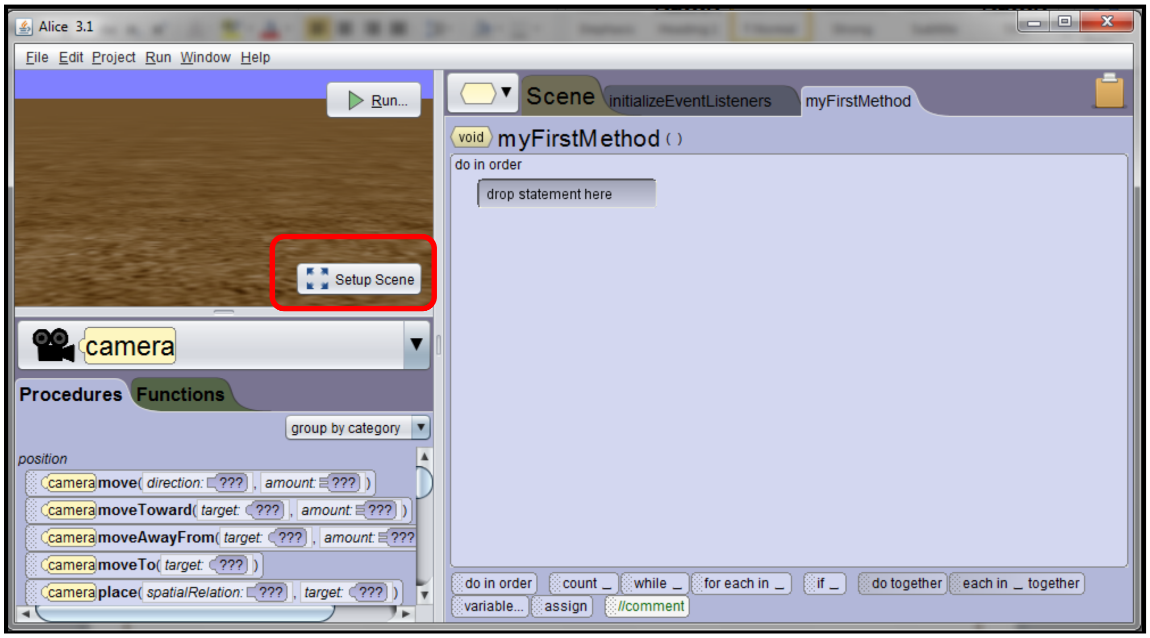
1. Open up Alice 3. You will need to find the installed Alice 3 folder and double click on the Alice 3 application file.

2. We are going to add a few objects to the Alice environment so that we can manipulate the objects and scene to get a better idea of how Alice works.

3. When you open the Alice program, you will be prompted to select a template. You also have the choice of selecting an old project. We will use this option at the end of this exercise. Make sure that the **Templates** tab is selected. Scroll down and select **Dirt**. Click **OK**.



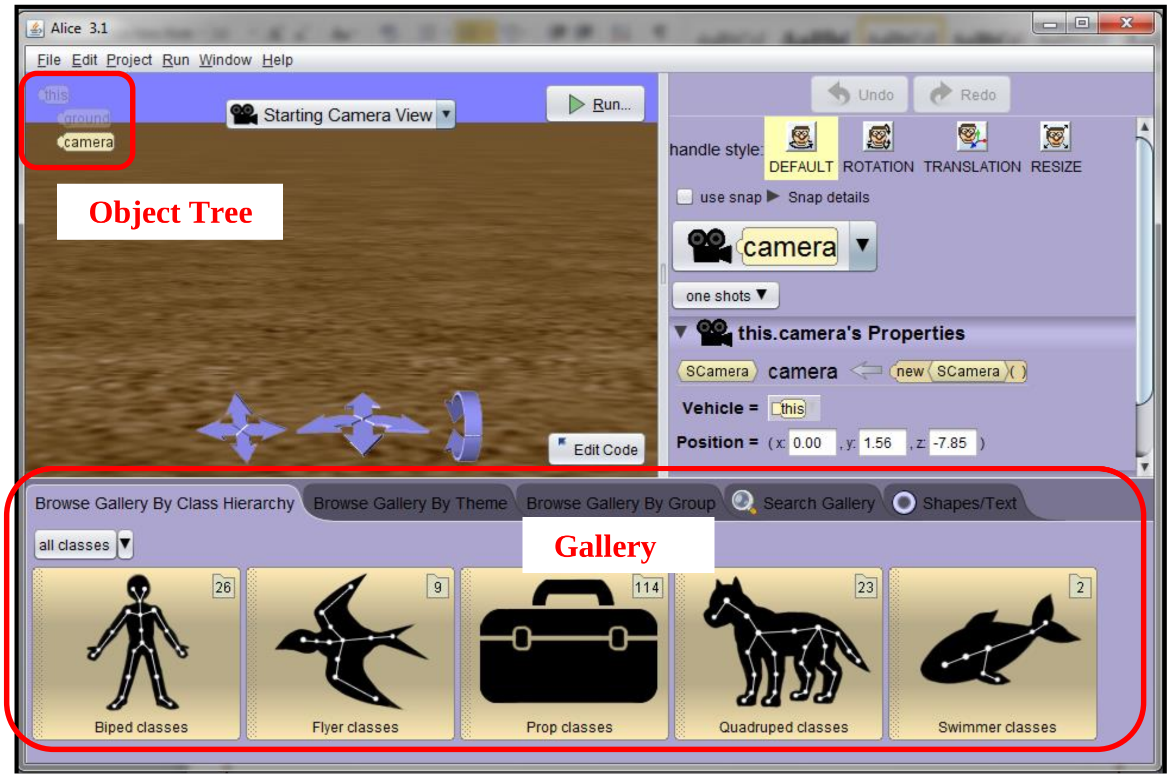
4. Your screen should look similar to the following. Before we can write any code, we need to add some objects to our scene. You will need to click on **Setup Scene** to add objects.



```
position
camera move( direction: ??? , amount: ??? )
camera moveToward( target: ??? , amount: ??? )
camera moveAwayFrom( target: ??? , amount: ???
camera moveTo( target: ??? )
camera place( spatialRelation: ??? , target: ??? )
```

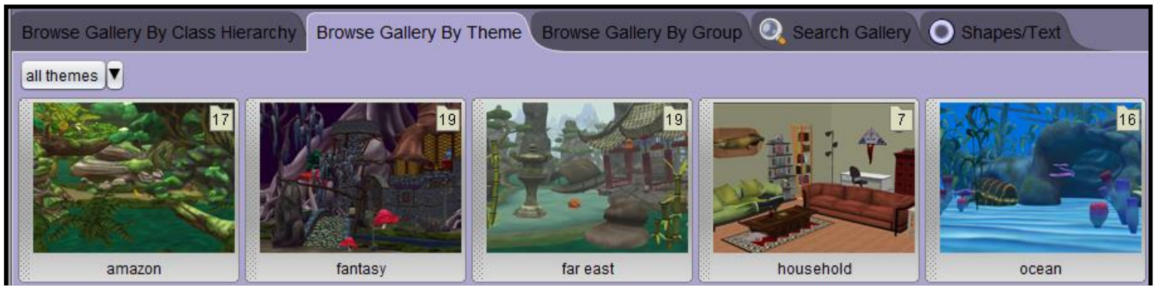
```
void myFirstMethod ( )
do in order
    drop statement here
```

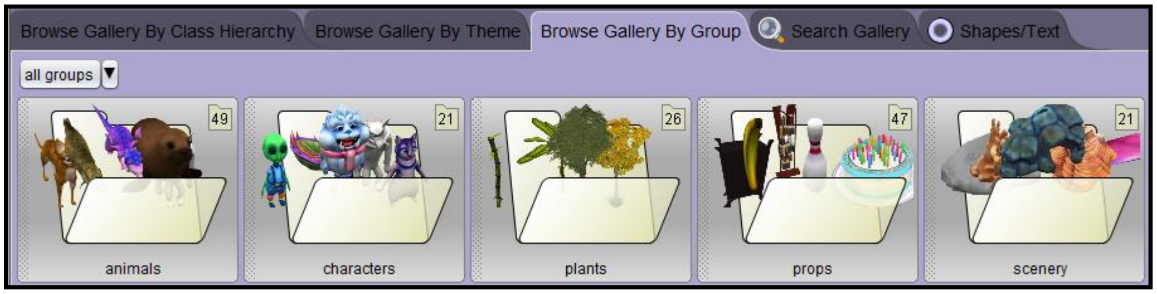
5. The scene setup area will look similar to the following. You currently have 2 objects in this scene: the camera and ground. You can see the objects in your scene by looking at the **object tree**. You can add objects to your scene by using the **gallery** options.



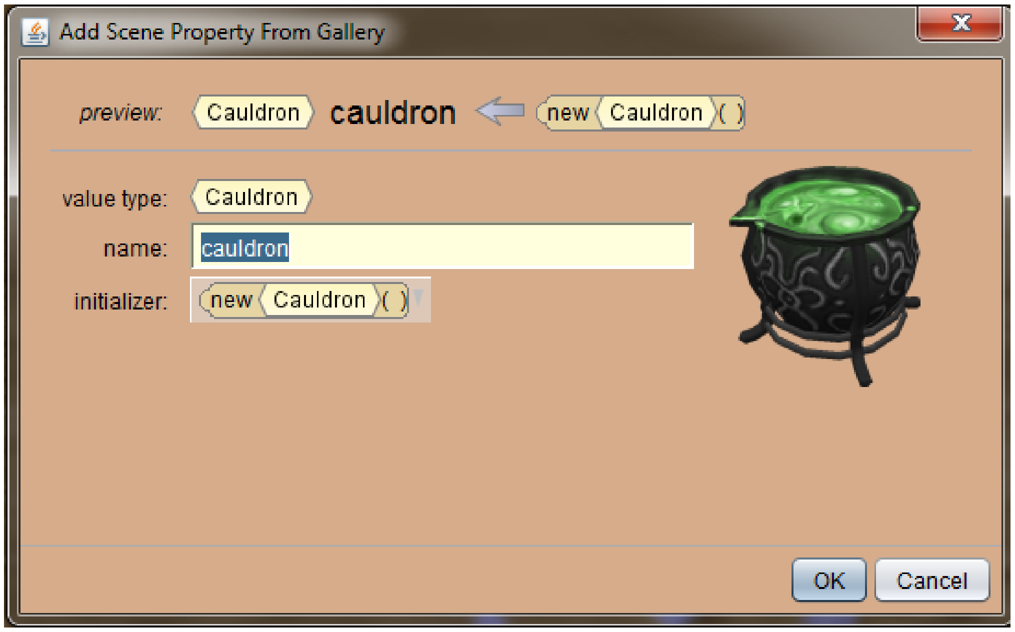
```
SCamera camera <- new SCamera ( )
Vehicle = this
Position = (x: 0.00 , y: 1.56 , z: -7.85 )
```

6. There are several choices for selecting objects from the gallery. Please take a moment to explore the possibilities.





7. Please click back on the **Browse Gallery by Class Hierarchy**. Then select **Prop Classes** and scroll until you find **Cauldron** (they are in alphabetical order). We are going to add this to our scene. Go ahead and click on it. You should be prompted to give this new object a name. We are going to leave the default name **cauldron** and click **OK**. We will talk about naming objects later in this chapter. Please leave the default names for this exercise.


Add Scene Property From Gallery

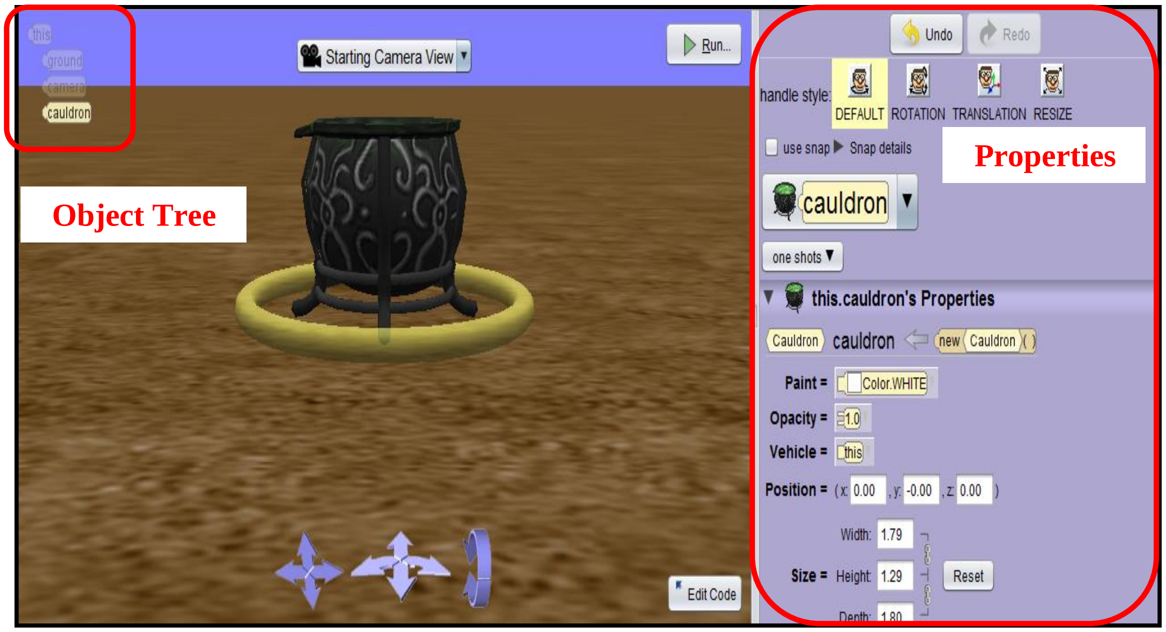
preview: Cauldron cauldron ⇐ new Cauldron ( )

value type: Cauldron

name: cauldron

initializer: new Cauldron ( )

8. You will notice that the cauldron appears in the center of the scene by default. It is also added to your object tree. The properties for the object are listed on the right pane as shown below. You can change the location, orientation, color, opacity, size, etc.



this.cauldron's Properties

Cauldron cauldron ⇐ new Cauldron ( )

Paint = Color.WHITE

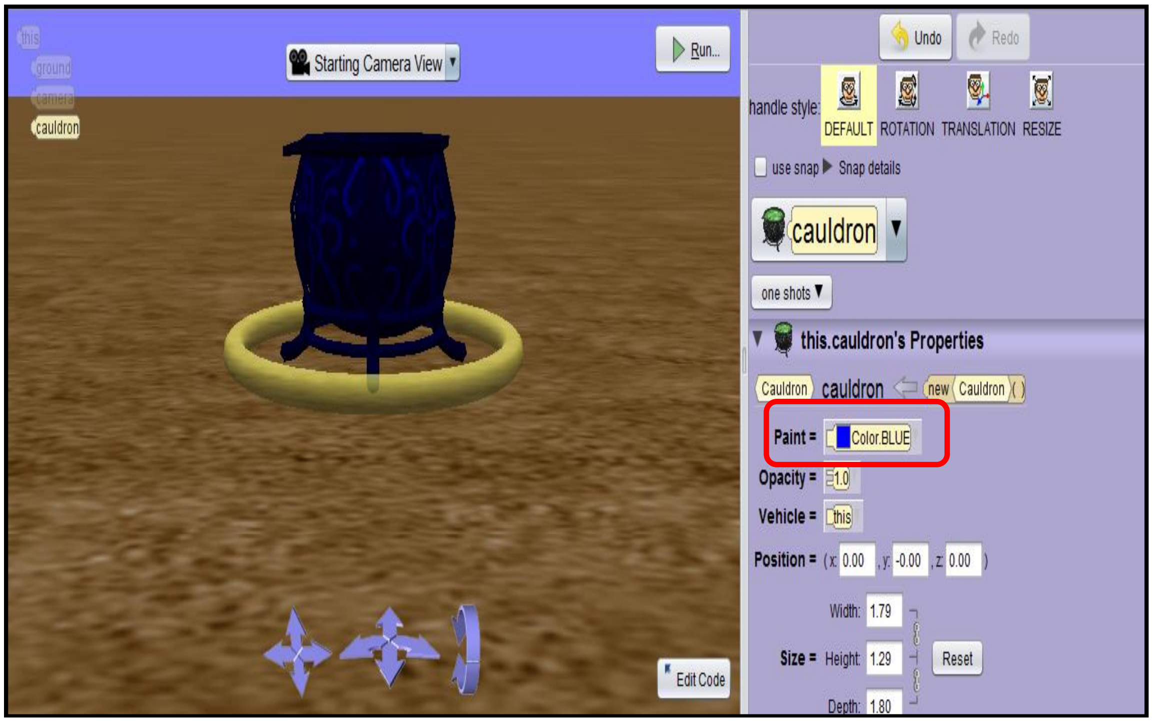
Opacity = 1.0

Vehicle = this

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

Size = Width: 1.79 Height: 1.29 Depth: 1.80 Reset

9. Let's change the color of the cauldron to blue. Click the drop down next to **Paint** and select **Color.BLUE**. You cauldron should change color.



this.cauldron's Properties

Cauldron cauldron ⇐ new Cauldron ( )

Paint = Color.BLUE

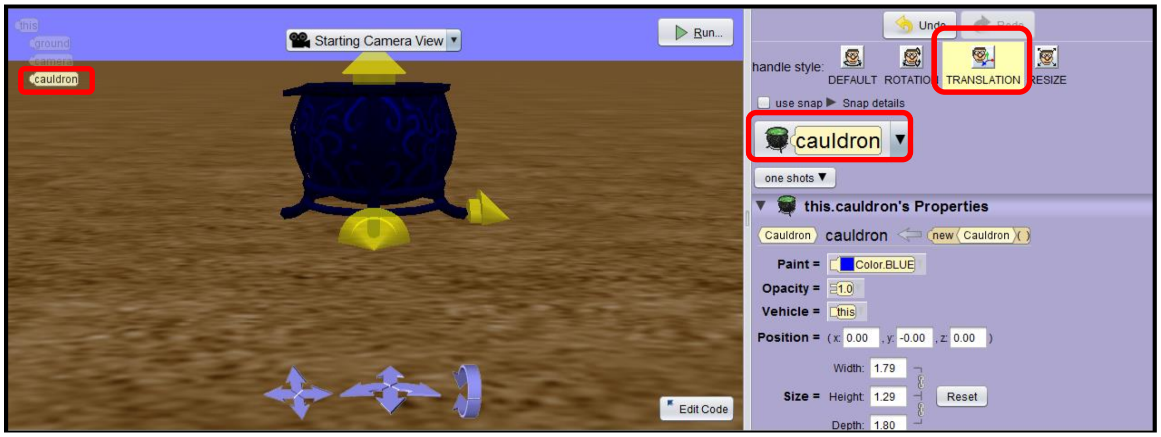
Opacity = 1.0

Vehicle = this

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

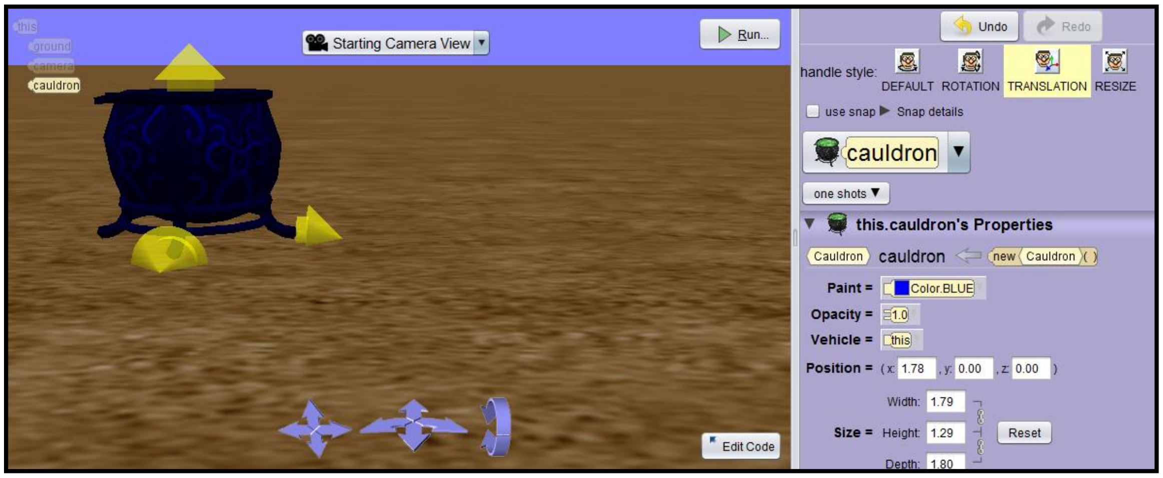
Size = Width: 1.79 Height: 1.29 Depth: 1.80 Reset

10. Next, we want to move the cauldron to the left side of the scene. The ring around the cauldron indicates that you can rotate the cauldron 360 degrees (this is the default option). I do not want to rotate the cauldron since it would look the same all the way around. I want to move it. To move an object, make sure that the object is selected from the object tree or from the drop down on the properties pane as shown below and select **Translation**.



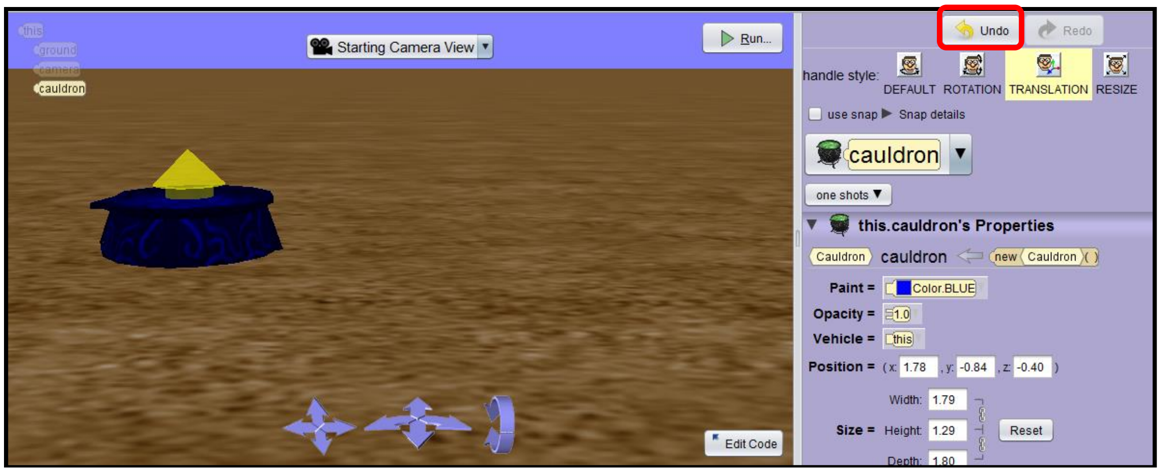
Paint = Color.BLUE
Opacity = 1.0
Vehicle = this
Position = (x: 0.00, y: -0.00, z: 0.00)
Size = Width: 1.79 Height: 1.29 Depth: 1.80

11. You will notice that the handle style changes to arrows. The arrow on top of the cauldron will move the cauldron up and down if you hold down your left mouse button on the arrow and drag up or down. The arrow to the right of the cauldron, will move the cauldron to the right or left. The front arrow will move the cauldron forwards and backwards. Try moving the cauldron to the left and forward.



Paint = Color.BLUE
Opacity = 1.0
Vehicle = this
Position = (x: 1.78, y: 0.00, z: 0.00)
Size = Width: 1.79 Height: 1.29 Depth: 1.80

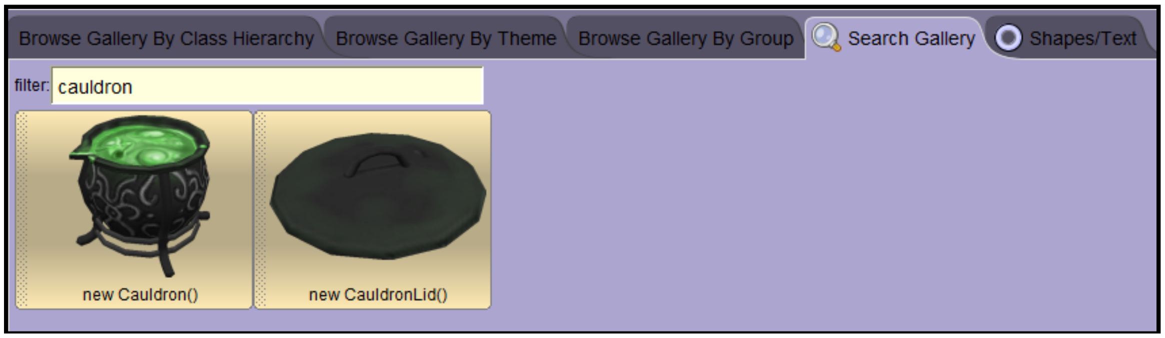
12. If you move the cauldron up or down, it will be off the ground. Go ahead and try it. You don’t have to worry about messing up the environment because you can undo. If your cauldron is in the ground or floating in the air, click the **undo** button as shown below.



Paint = Color.BLUE
Opacity = 1.0
Vehicle = this
Position = (x: 1.78 , y: -0.84 , z: -0.40)
Size = Width: 1.79 Height: 1.29 Depth: 1.80

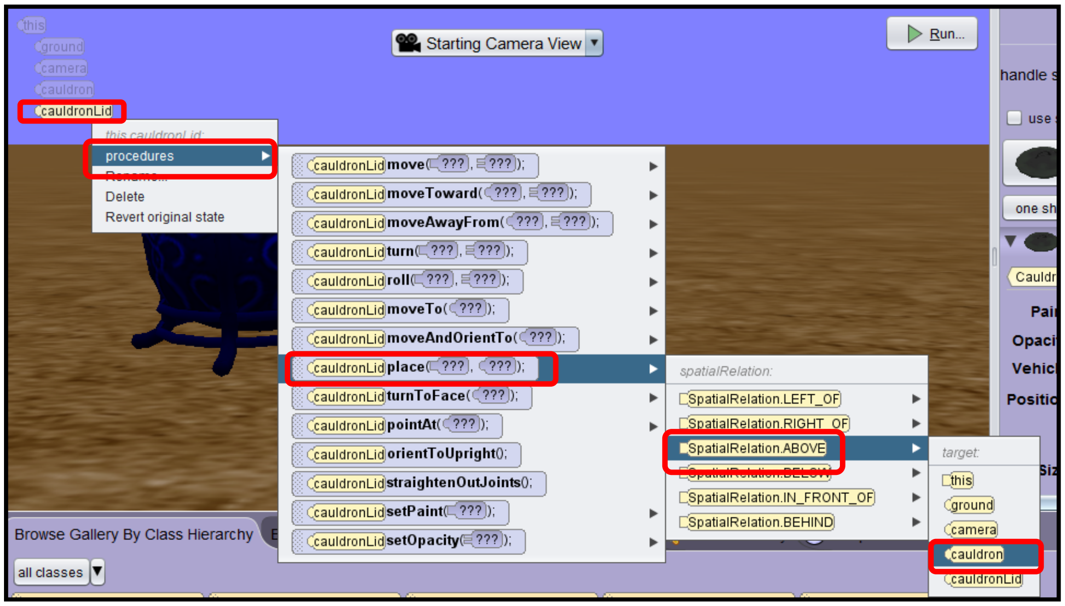
13. Do not resize the cauldron. We are going to add the cauldron lid and if we resize the cauldon, it won’t fit and we will have to resize the cauldron lid to fit. We will practice with rotating and resizing in the next exercise.

14. Next, let’s add the cauldron lid. See if you can find it. You can try using the search since we know the object that we want to add. Click on the CauldronLid to add a cauldron lid to your scene. Leave the default name for the object.

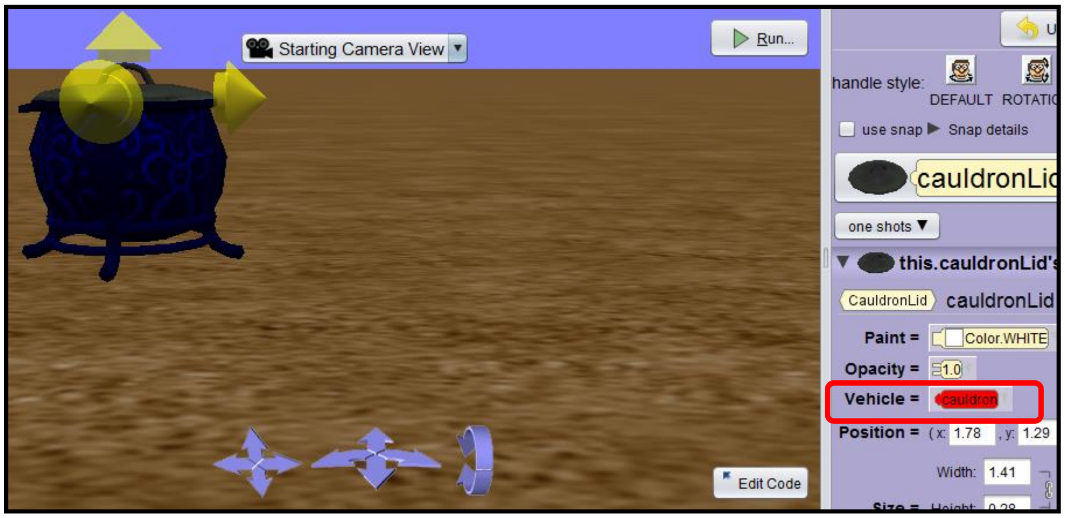


filter: cauldron

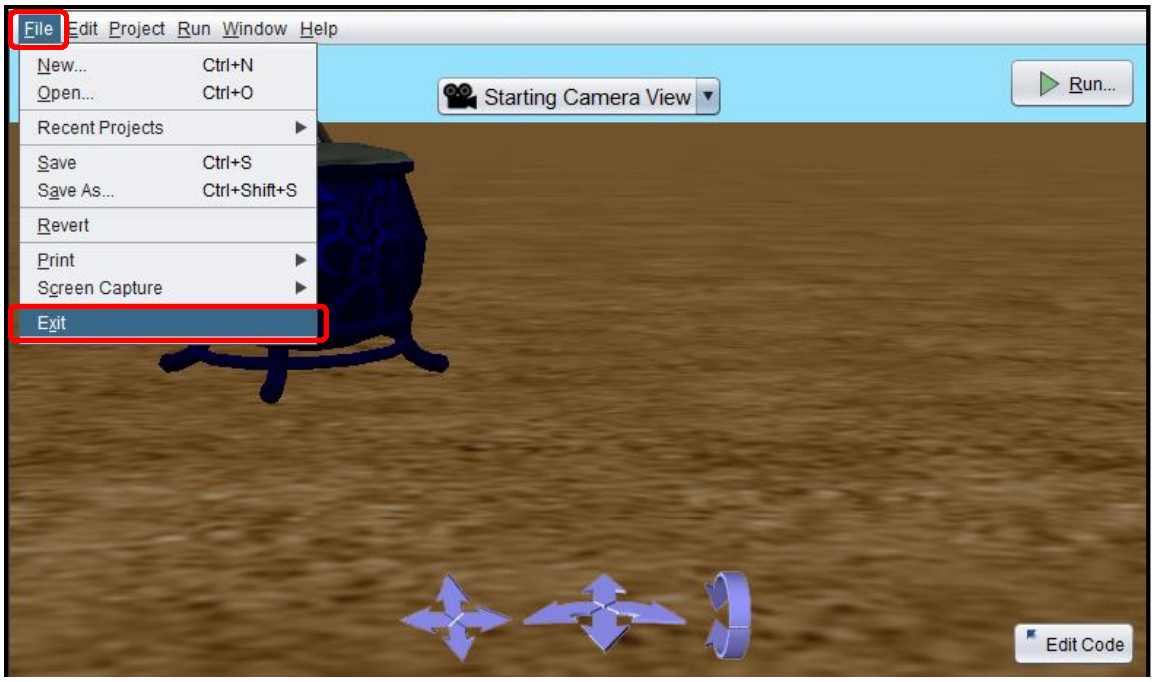
15. The cauldron lid is added to our scene, but it is on the ground. We could move the cauldron up, to the left, and forward to get it on top of the cauldron, but there is an easier way. There is a One Shot drop down that will allow us to move the cauldron lid onto the cauldron without having to move it ourselves. Make sure the cauldron lid is selected and right click on the cauldron lid from the object tree. Choose **procedures**, **place**, **above**, and select **cauldron**. Be careful with the cascading menus when you are selecting options. You can think of procedures as actions.


this
ground
camera
cauldron
cauldronLid
Starting Camera View
Run...
this.cauldronLid:
procedures
Rename...
Delete
Revert original state
cauldronLid move(???, ???);
cauldronLid moveToward(???, ???);
cauldronLid moveAwayFrom(???, ???);
cauldronLid turn(???, ???);
cauldronLid roll(???, ???);
cauldronLid moveTo(???);
cauldronLid moveAndOrientTo(???);
cauldronLid place(???, ???);
cauldronLid turnToFace(???);
cauldronLid pointAt(???);
cauldronLid orientToUpright();
cauldronLid straightenOutJoints();
cauldronLid setPaint(???);
cauldronLid setOpacity(???);
spatialRelation:
SpatialRelation.LEFT_OF
SpatialRelation.RIGHT_OF
SpatialRelation.ABOVE
SpatialRelation.BELOW
SpatialRelation.IN_FRONT_OF
SpatialRelation.BEHIND
target:
this
ground
camera
cauldron
cauldronLid
Browse Gallery By Class Hierarchy
all classes

16. The cauldron lid should now be on top of the cauldron, but if we move the cauldron the lid will not move with it. To make the lid move with the cauldron, we need to change the vehicle property of the lid to the cauldron. Make sure that the cauldron lid is selected and change the **vehicle property** (makes one object move in conjunction with the other object) to **cauldron** as shown below. Try moving the cauldron, does the lid move with it?



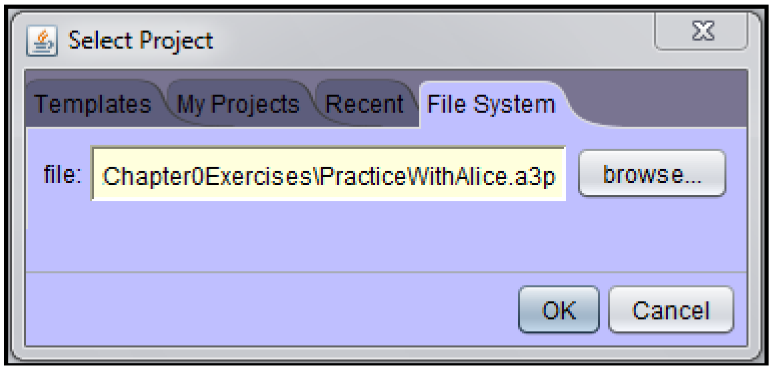
17. We are going to save our project and continue our work in next exercise. To save the project, you should click on **File**, **Save As**. Name the project **PracticeWithAlice**. Please get in the habit of capitalizing the first letter of every word in your filename and do not use space when naming your files. You should save this file in your **Chapter0Exercises** folder. Exit Alice by clicking **File, Exit.**



Exercise 2: Manipulating the Alice Environment – Part 2

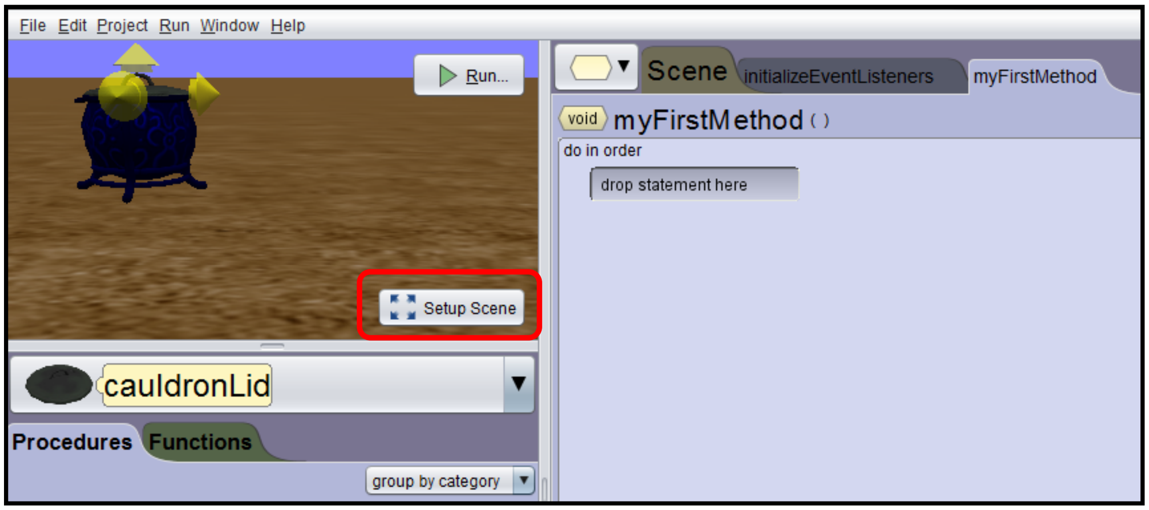
1. Open up Alice 3. You will need to find the installed Alice 3 folder and double click on the Alice 3 application file.

2. Click on the **File System** tab and choose **browse…** You will need to locate your **PracticeWithAlice** file and click **OK**. *(Note: Alice files have an .a3p file extension. If you double click on this file, it will not open in Alice. Unless you create a file association, you will need to open all of your Alice projects from the Alice software.)*

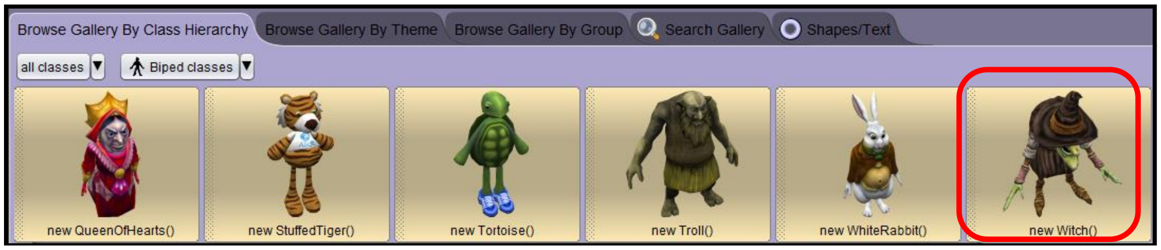


file: Chapter0Exercises\PracticeWithAlice.a3p

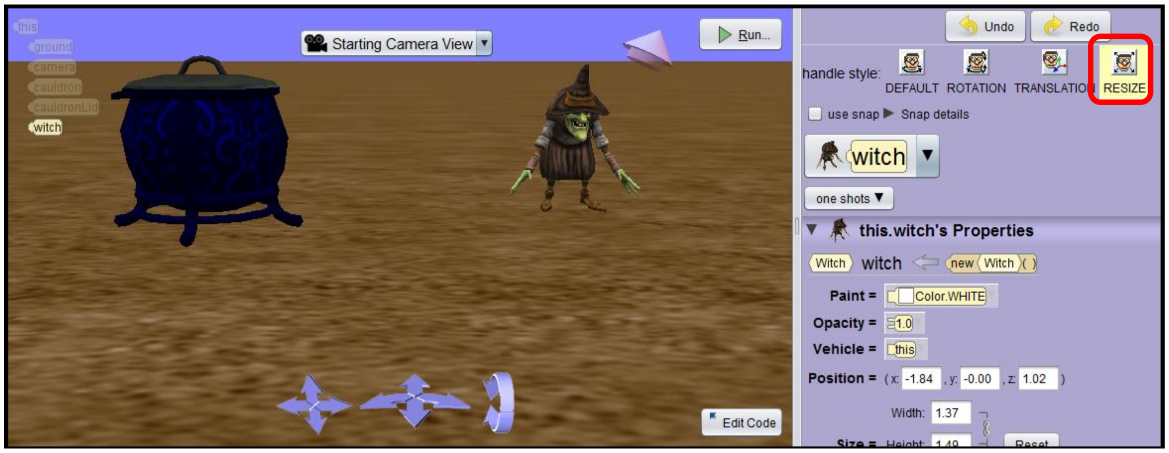
3. You should have a cauldron and a cauldron lid in your scene from exercise 1. Click **Scene Setup** so that we can practice some more with manipulating objects.



4. Let's add a **witch** to the scene. There are multiple ways to find her. You can search for her, click on hierarchy and biped class, click on theme and fantasy, or click on group and characters. Instead of clicking on the witch to add her to the scene, hold down your left mouse button on the **new Witch()** and drag onto the scene and release where you want her. You will notice a yellow bounding box on the screen indicating where she is going to appear. Leave the default name for her. If this method of adding objects does not work for you, then you can add the object by clicking on it and then moving it once it is added to the scene.



5. Make sure that the **witch** is selected on your scene and then click on **resize**. You should notice an arrow above the witch's head. This will resize the witch proportionally if you hold down your left mouse button and drag up or down. Make her bigger. Whatever you think looks good.



Witch witch ⇐ new Witch ( )

Paint = Color.WHITE

Opacity = 1.0

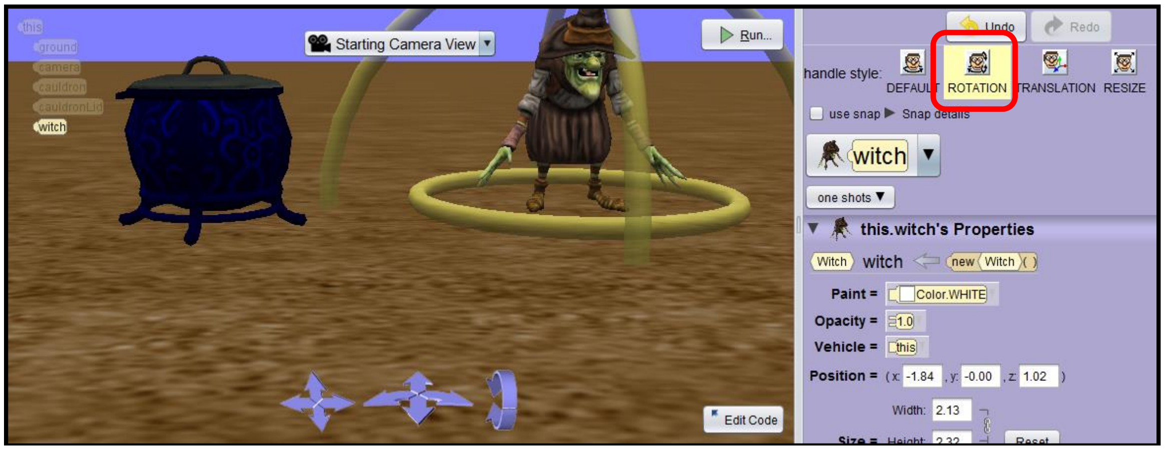
Vehicle = this

Position = (x: -1.84 , y: -0.00 , z: 1.02)

Width: 1.37

Size = Height: 1.49

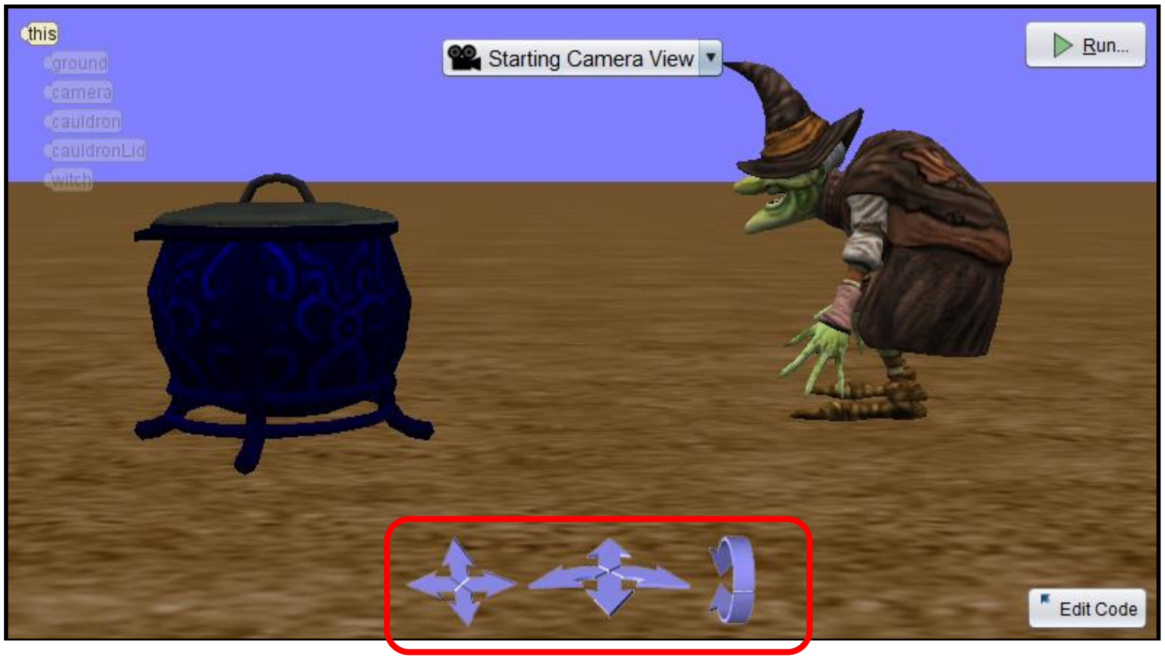
6. Now, let's rotate her. Click on **rotation**. The up and down yellow lines will rotate the witch forwards and backwards if you hold down your left mouse button and drag. The yellow circle on the bottom of the witch will rotate the witch 360 degrees. Hold down your left mouse button on the bottom circle and drag to the left. Rotate the witch so that she is facing the cauldron.



Witch witch ⇐ new Witch ( )

Paint = Color.WHITE
Opacity = 1.0
Vehicle = this
Position = (x: -1.84 , y: -0.00 , z: 1.02)
Width: 2.13
Size = Height: 2.32

7. Now, we are going to practice with the camera movements. The first set of arrows on the left will move the scene up, down, left, and right. The second set of arrows will move the witch forwards and backwards. The last set of arrows will adjust the scene up and down (more or less sky). Please try out each of the camera arrows.

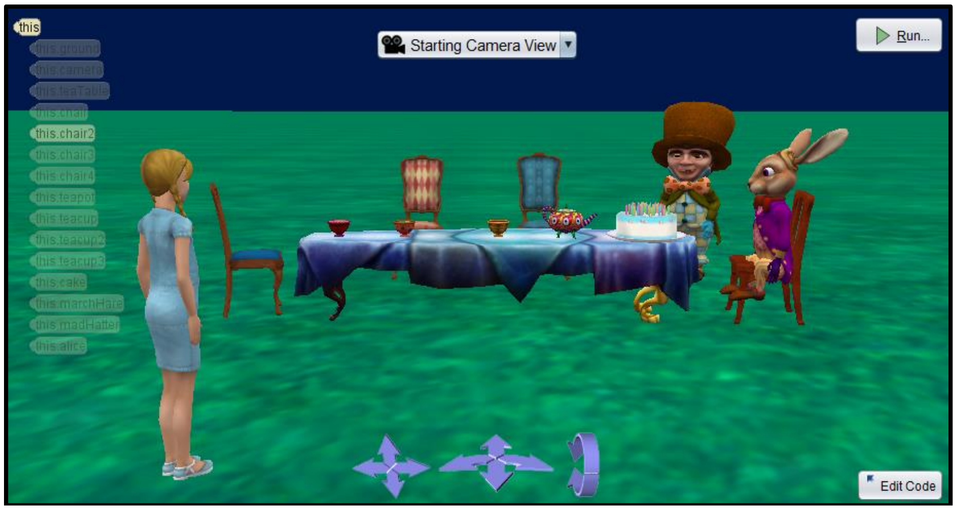


8. Please take some time to add more objects and manipulate those objects. Practicing with the environemnt is the best way to get acquainted with it. Save your work.

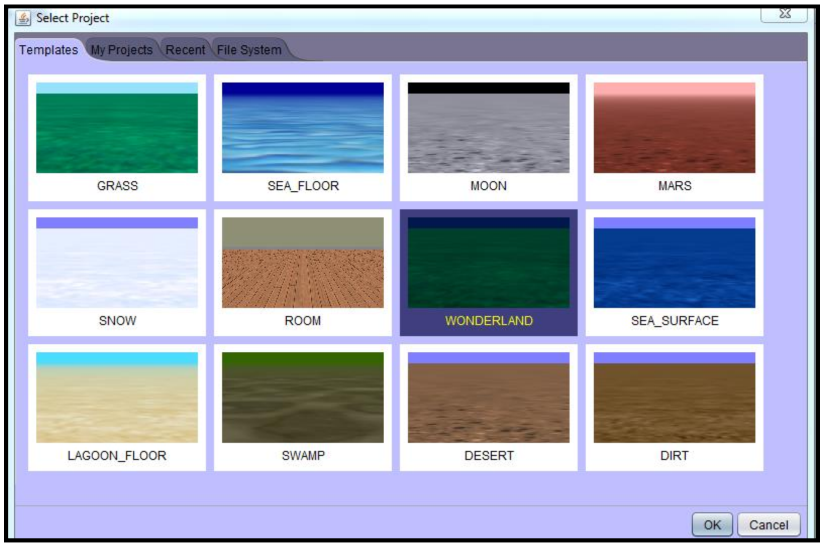
Exercise 3: Alice in Wonderland Mad Tea Party Scene Setup (ongoing exercise)

1. Open up Alice 3. You will need to find the installed Alice 3 folder and double click on the **Alice 3** application file. The first step in programming is understanding the problem. We would like to create a trimmed version of the Alice in Wonderland unbirthday tea party. Once you understand the problem, you setup the scene and create a storyboard for animating the scene.

2. Our goal for the scene setup is to have the following characters: Alice, Mad Hatter, and the March Hare. We will also add some objects to make the scene more interesting: a table, chairs, a tea pot, tea cups, and a birthday cake. When we are finished it should look similar to the following:

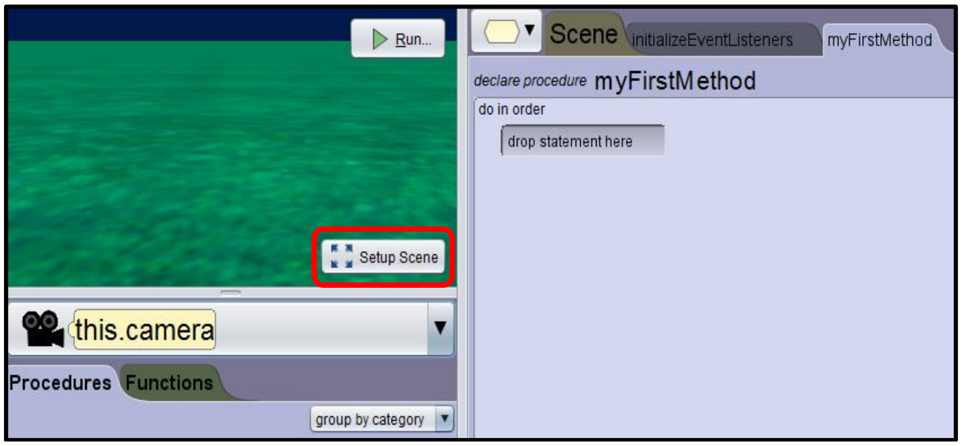


3. Select the **wonderland** template:

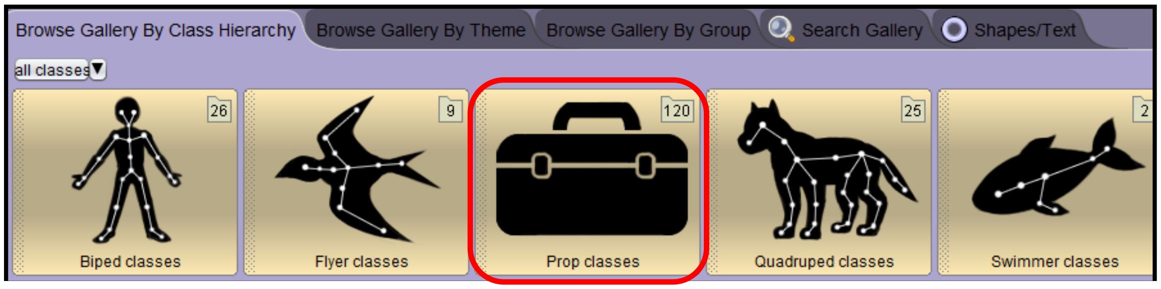


4. Select **File** from the menu, then **Save As**. Save this file as **TeaParty**. Please get in the habit of capitalizing the first letter of every word in your filename and do not use spaces when naming your files. You should save your work often. You can click Save from the File menu from this point on. You should save this file in your **Chapter0Exercises** folder.

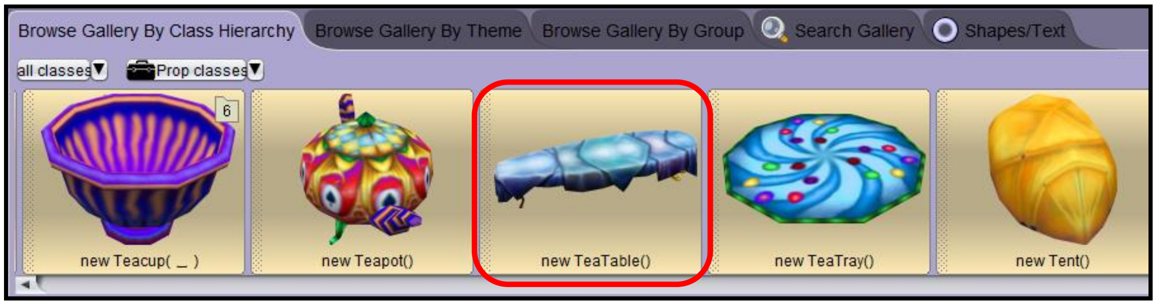
5. Click on **Setup Scene button.**



6. We are going to add a table to the scene for the characters to gather around. There is a tea table specifically designed for Alice in Wonderland. Click on the tab called **Browse Gallery By Class Hierarchy**. Click on the **Prop classes** category.

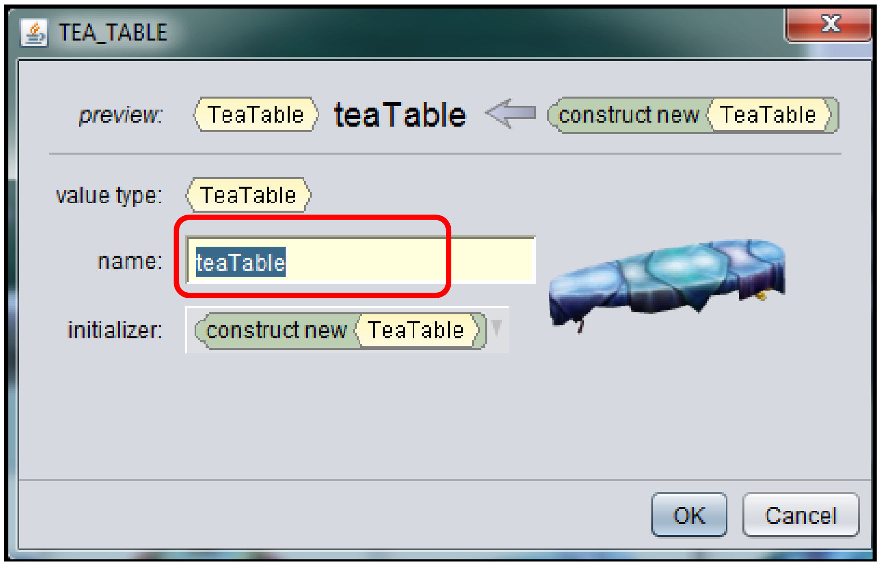


7. Scroll to the end (they are in alphabetical order) until you see the **TeaTable** class. You could have used the ***Search Gallery*** tab to find the table as well.



8. Click on the **TeaTable** class to add a tea table to your world or hold down your left mouse button and drag this object to wherever you would like to place it in your scene. If you choose to click on the TeaTable class, the new object will be placed automatically in the center of the scene.

9. When you click on the class it will ask you for a name for the object. You can leave the name teaTable or rename if you want. Do not put spaces in your object name and the first letter of your object name should begin with a lowercase letter and the first letter of the second word should be a capital letter. Object names begin with a lowercase letter; we will talk more about this in the next chapter.


TEA_TABLE

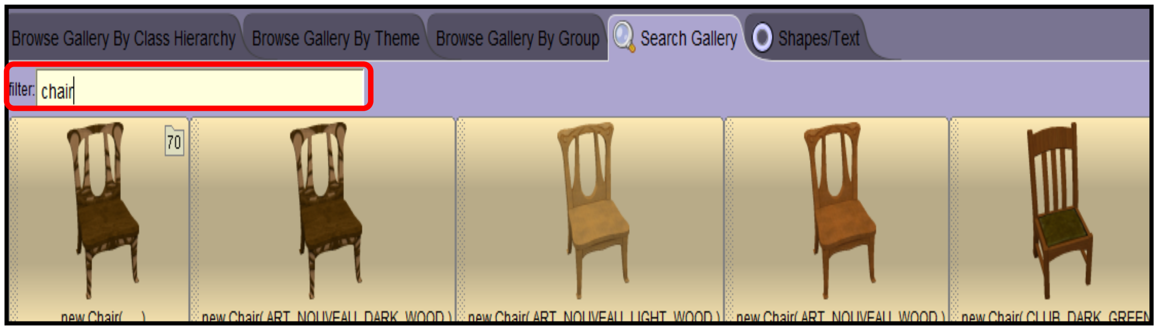
preview: TeaTable teaTable ⟸ construct new TeaTable

value type: TeaTable

name: teaTable

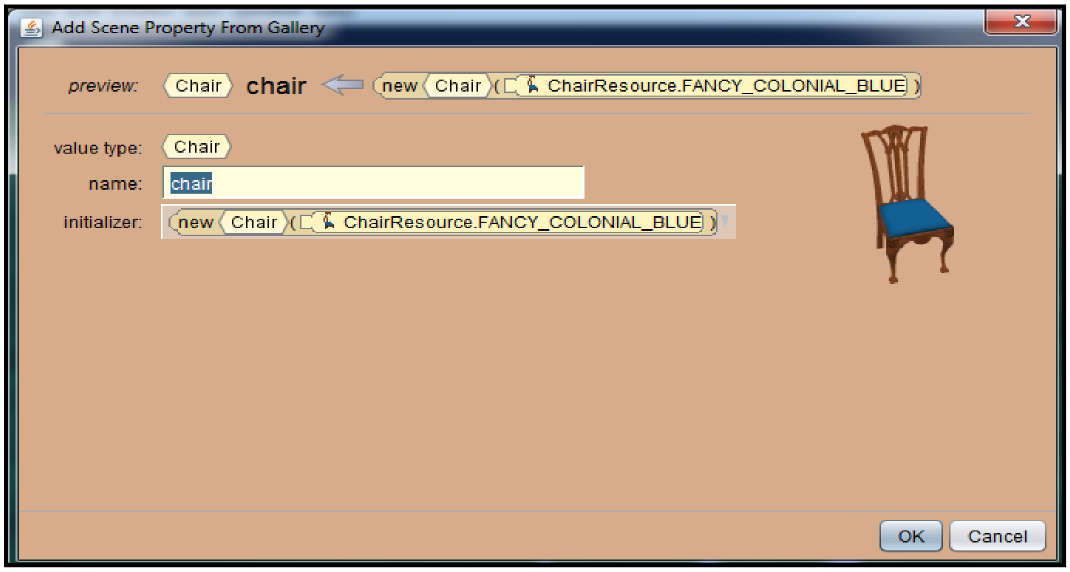
initializer: construct new TeaTable

10. Next, we are going to **add a chair**. Now we can test out the search feature in Alice by typing chair into the search box. You will have a list of all the chair models. Please choose the chair that you like.


Browse Gallery By Class Hierarchy
Browse Gallery By Theme
Browse Gallery By Group
Search Gallery
Shapes/Text
70
new Chair(...)
new Chair(ART_NOUVEAU_DARK_WOOD)
new Chair(ART_NOUVEAU_LIGHT_WOOD)
new Chair(ART_NOUVEAU_WOOD)
new Chair(CLUB_DARK_GREEN

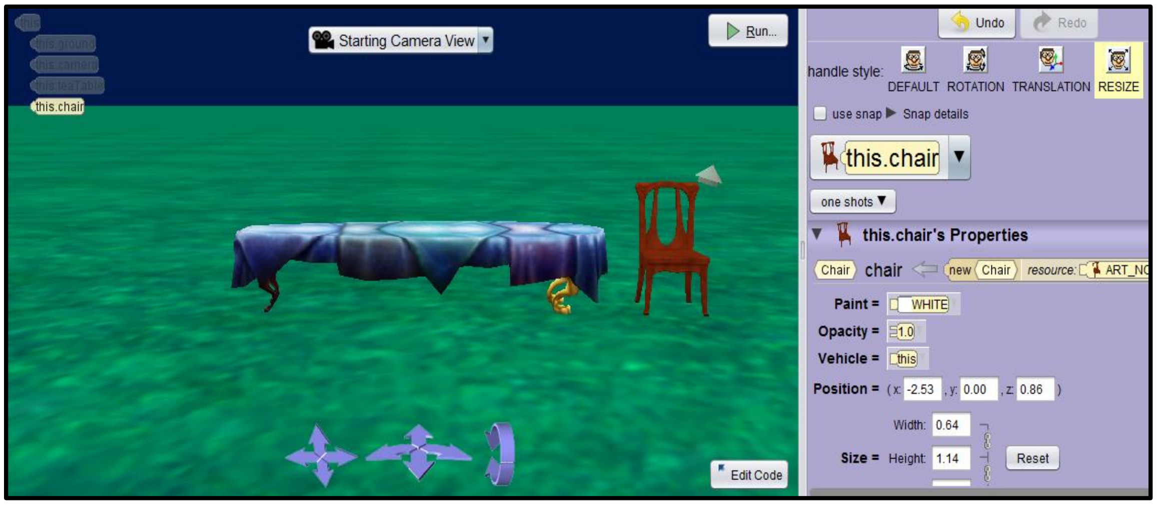
filter: chair

11. Drag the chair that you want onto the scene where you want it by holding down the left mouse button and dragging from the class that you are choosing to add. You will see a yellow bounding box that shows you were your new object will be placed. When you get the object where you want it, release and it will ask you for a name for the object.

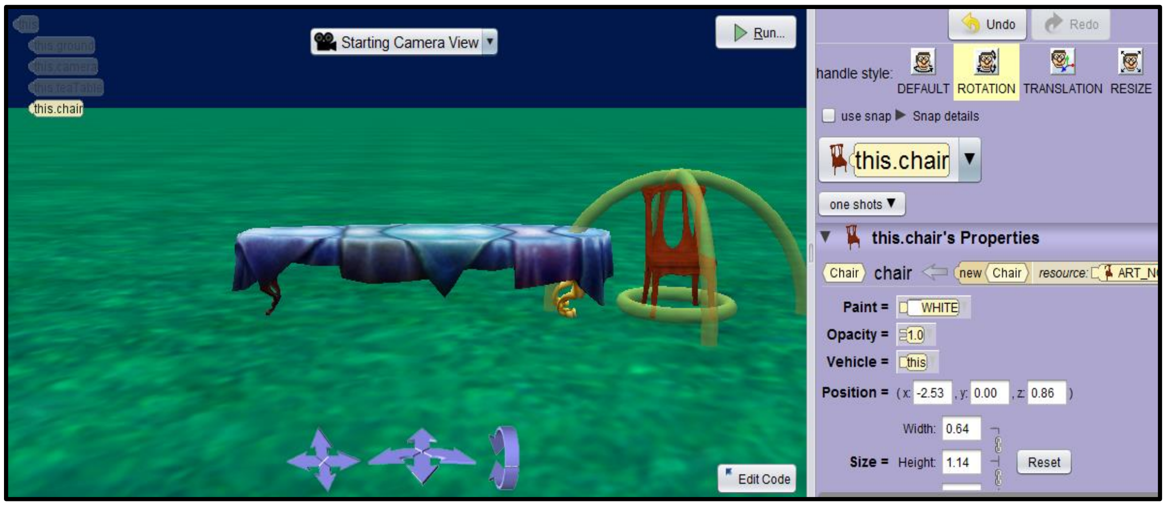

Add Scene Property From Gallery
preview: Chair chair ⇐ new Chair(ChairResource.FANCY_COLONIAL_BLUE)
value type: Chair
name: chair
initializer: new Chair(ChairResource.FANCY_COLONIAL_BLUE)
OK
Cancel

You should name this object something simple. Let's name it **chair**.

12. We should resize the chair so that it matches the size of the table. To do this, you will need to select the chair and then click on the **resize** button from the handle style choices. When you click on the button, an arrow will appear above the chair. Holding down your left mouse button on the arrow and move your mouse up and down to resize the chair.



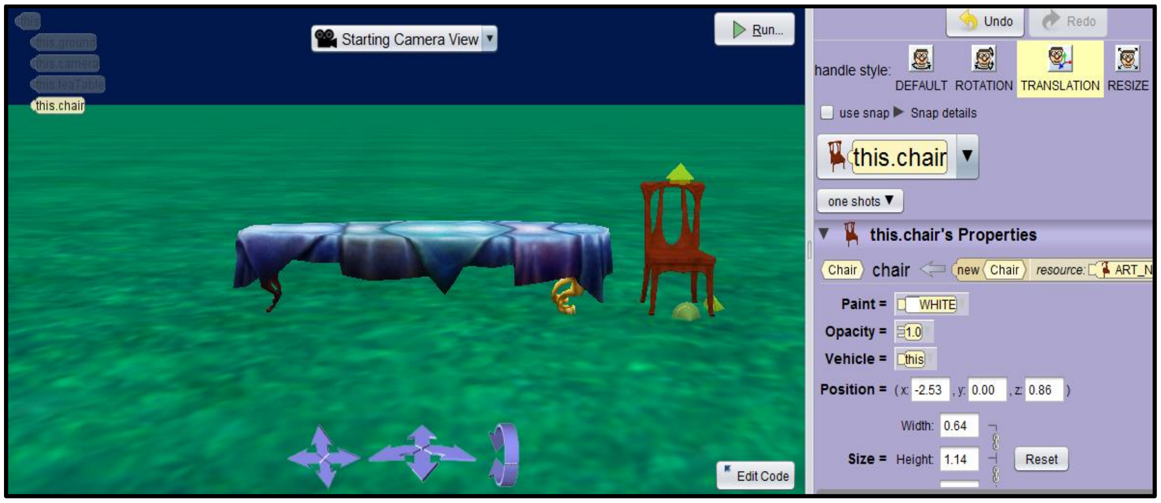
13. To rotate the chair, click on the **rotation** button from the handle style choices. If you hold down the left mouse button on the bottom ring and drag to the right and left, it will spin the chair around so that you can have it face the table.



Chair chair ⇐ new Chair resource: ART_N

Paint = WHITE
Opacity = 1.0
Vehicle = this
Position = (x: -2.53 , y: 0.00 , z: 0.86)
Size = Width: 0.64 Height: 1.14

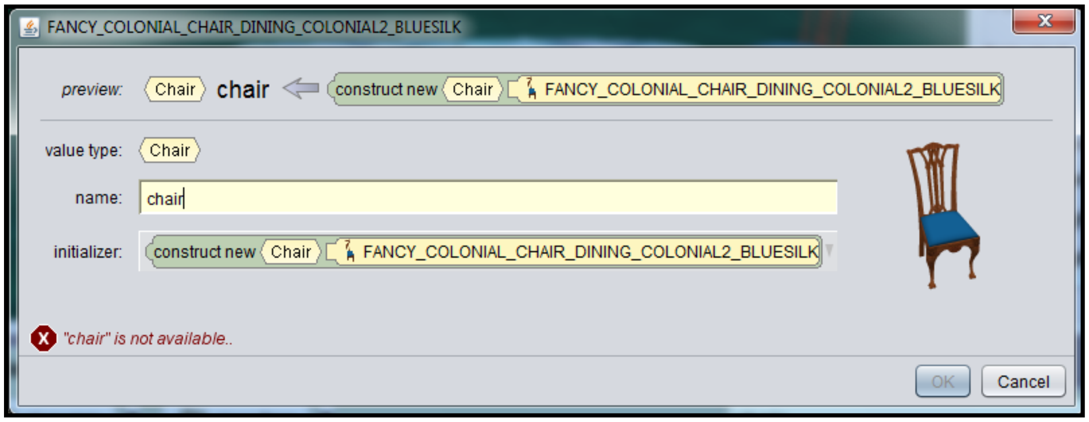
14. To move the chair, click on the **translation** button from the handle style choices. If you hold down your left mouse button on the arrow on top of the chair and drag up and down, the chair will move up and down. The arrow in the front will move the chair forward and backward. The arrow to the right will move the chair left and right.



Chair chair ⇐ new Chair resource: ART_N

Paint = WHITE
Opacity = 1.0
Vehicle = this
Position = (x: -2.53 , y: 0.00 , z: 0.86)
Size = Width: 0.64 Height: 1.14

15. **Add 3 more chairs** to the scene around the table. Be careful not to give the chairs the same name. You will see the following error if you try to name your objects the same name. You can call the other chairs: chair2, chair3, and chair4. Do not put spaces in your names. The Alice software will not allow you to name your objects with spaces and this is because the Java language does not allow you to have spaces when naming.


FANCY_COLONIAL_CHAIR_DINING_COLONIAL2_BLUESILK

preview: Chair chair ⇐ construct new Chair FANCY_COLONIAL_CHAIR_DINING_COLONIAL2_BLUESILK

value type: Chair

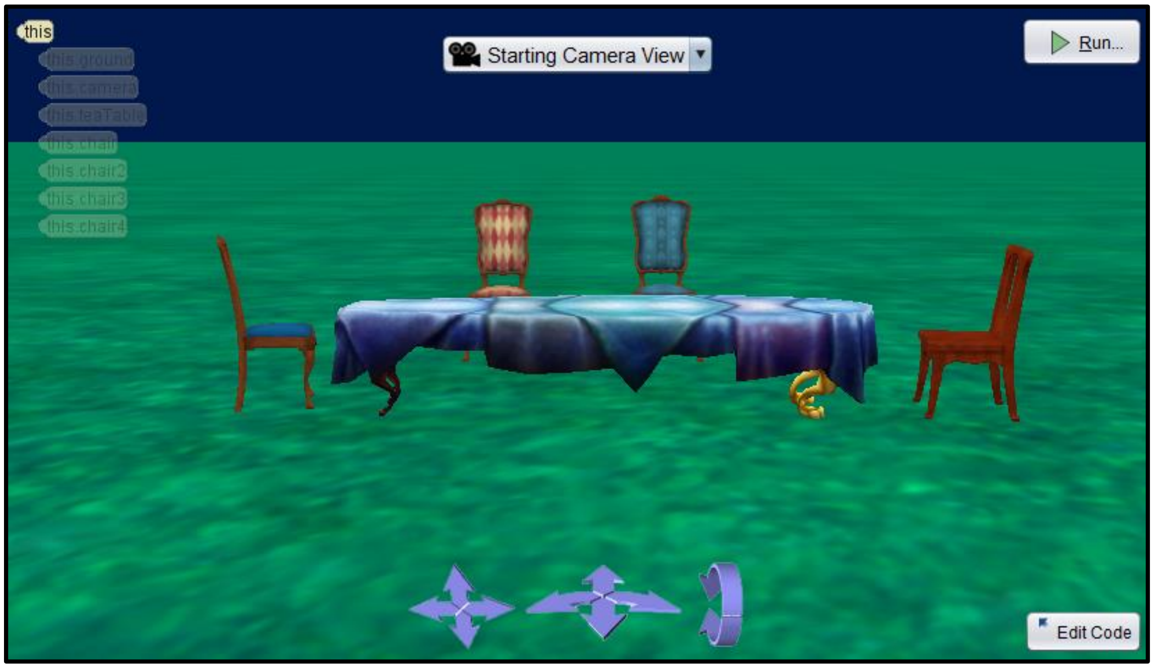
name: chair

initializer: construct new Chair FANCY_COLONIAL_CHAIR_DINING_COLONIAL2_BLUESILK

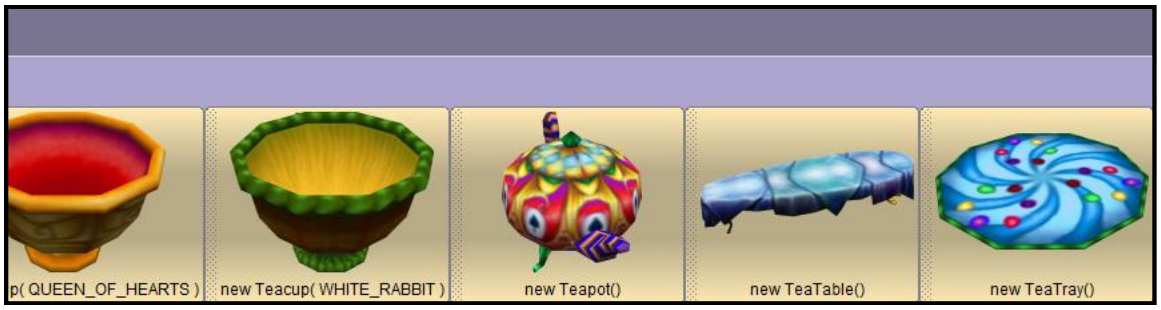
"chair" is not available..

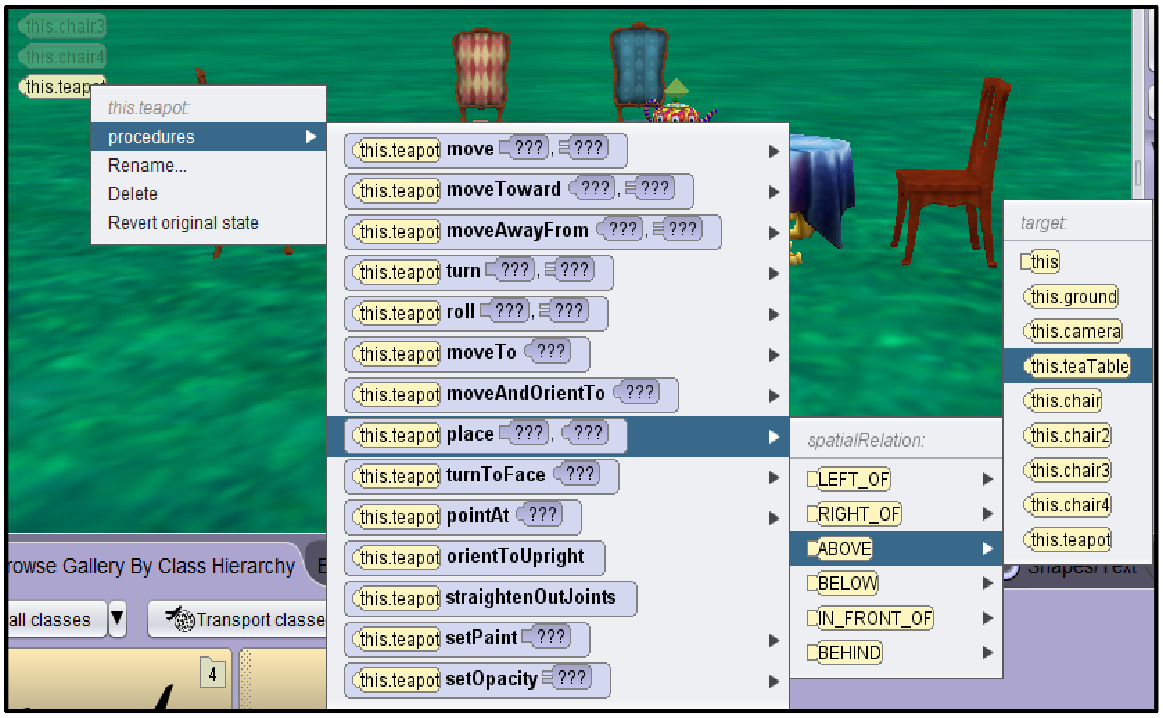
OK Cancel

16. It should look similar to the following:



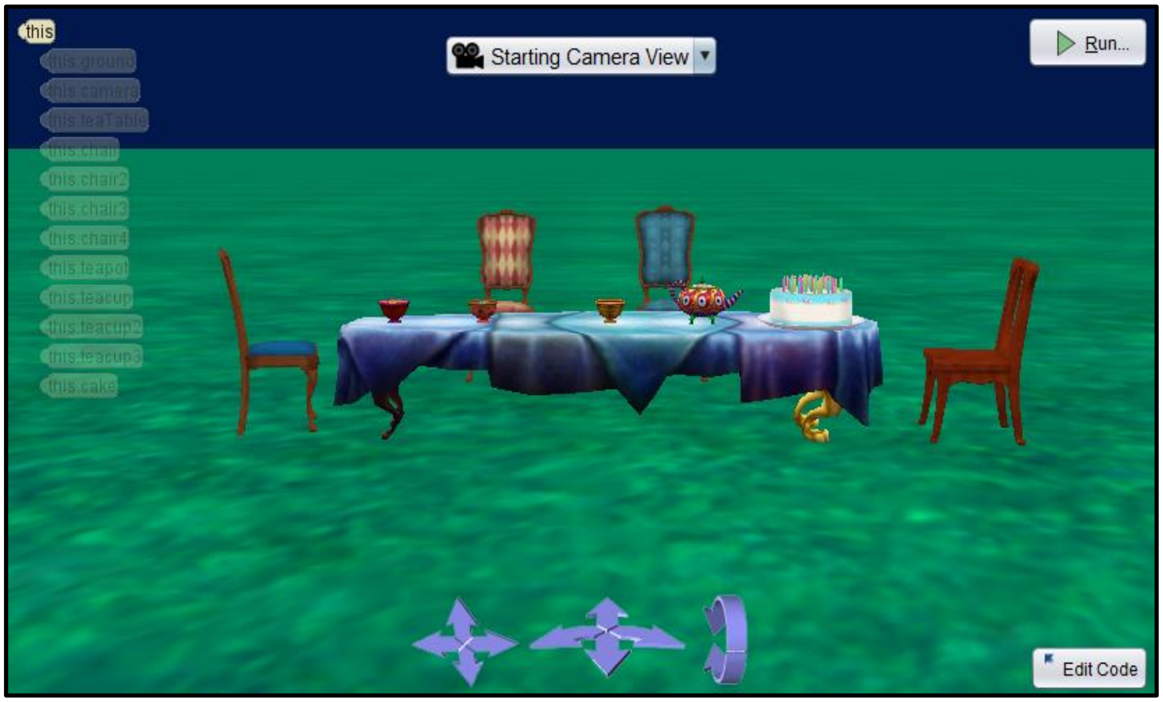
17. Next, we need to **add some teacups and a teapot onto the table**. If you search for tea in the gallery, you will be given the teapot, teacups, saucers, etc. I would like to start with the teapot. When you create the teapot, you can use the default name. We can play with trying to get this teapot onto the table, but this would take a while and there is an easier way. If you **right click on the teapot**, select **procedures**, **teapot place…**, **above**, and **teaTable**, it will place the teapot on top of the table for you.



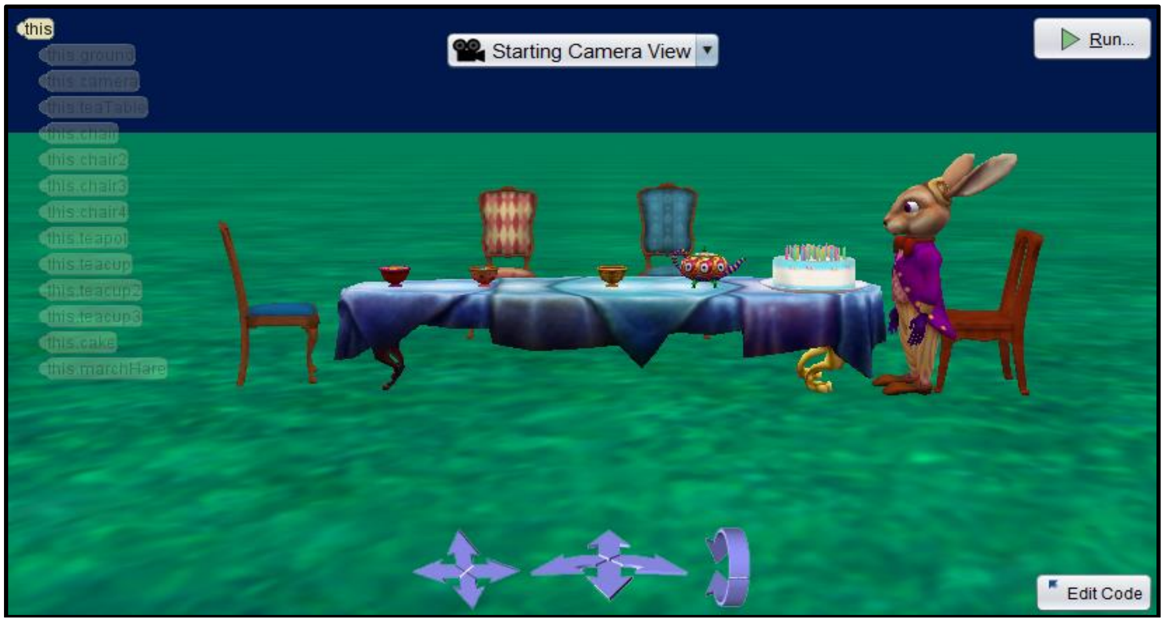


18. **Add a few teacups onto the table** and adjust them how you want them. *Be careful not to give 2 teacups the same name.*

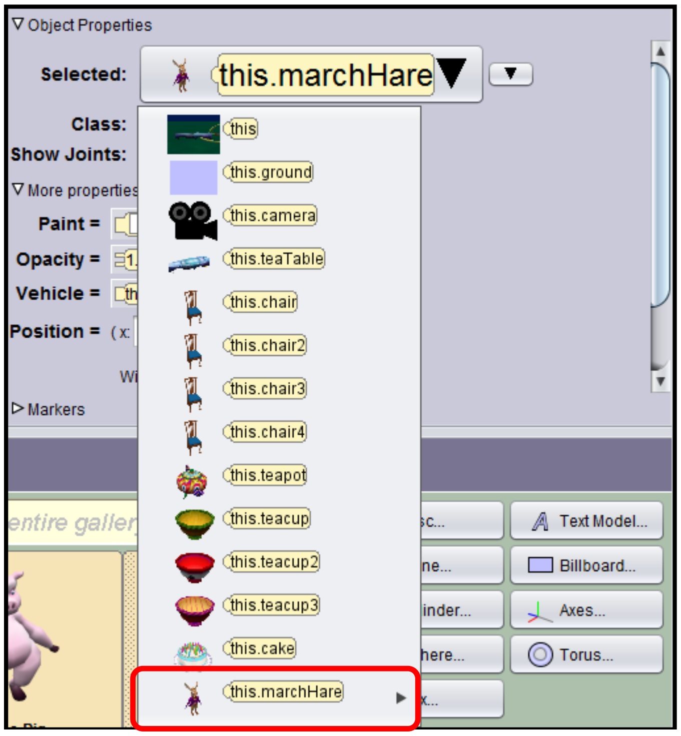
19. **Add a birthday cake onto the table** and readjust the items on the table. It should look similar to the following.

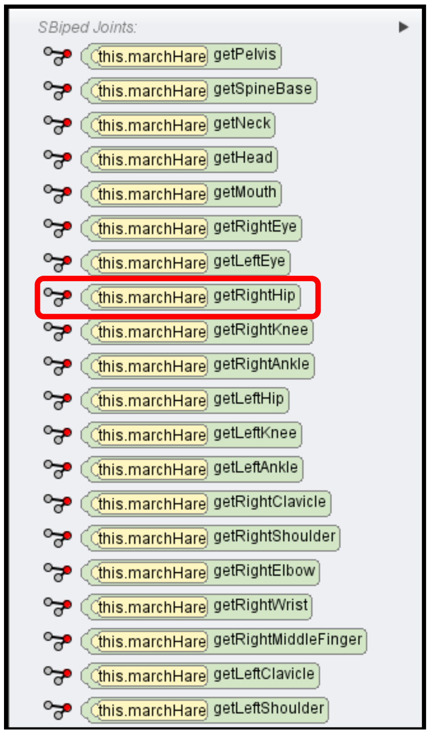


20. Next, we are going to add the characters. The characters can be found in the biped folder in the gallery. Let's add the March Hare first. Place him directly in front of one of the chairs. It doesn't matter which chair you choose. You will need to rotate him so that he lined up with the chair. We are going to make him sit in the chair.

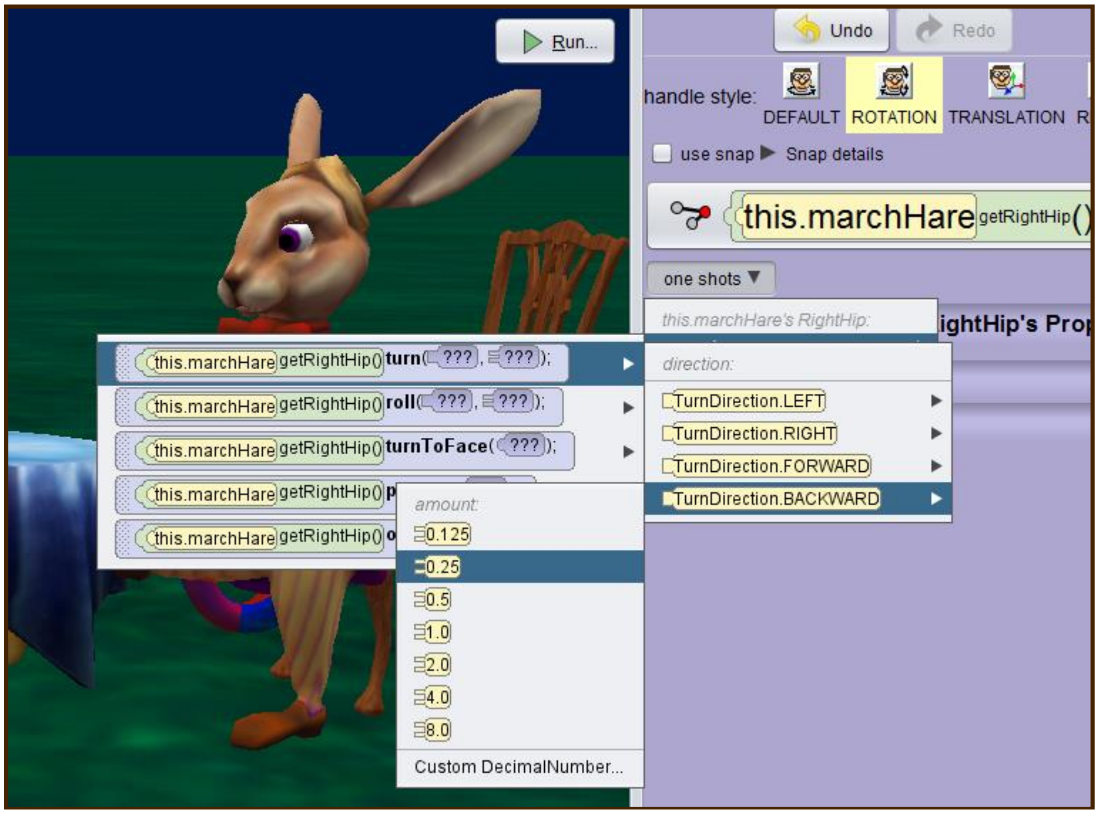


21. To move the marchHare's joints, we will need to select the marchHare and drop down his subparts as shown below. Choose the hare's right hip.




SBiped Joints:
this.marchHare getPelvis
this.marchHare getSpineBase
this.marchHare getNeck
this.marchHare getHead
this.marchHare getMouth
this.marchHare getRightEye
this.marchHare getLeftEye
this.marchHare getRightHip
this.marchHare getRightKnee
this.marchHare getRightAnkle
this.marchHare getLeftHip
this.marchHare getLeftKnee
this.marchHare getLeftAnkle
this.marchHare getRightClavicle
this.marchHare getRightShoulder
this.marchHare getRightElbow
this.marchHare getRightWrist
this.marchHare getRightMiddleFinger
this.marchHare getLeftClavicle
this.marchHare getLeftShoulder

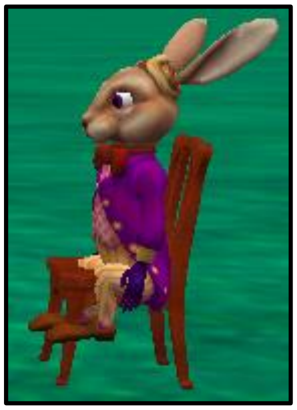
Now, we need to select **ONE SHOT, procedures**, **marchHare.getRightHip.turn…**, **BACKWARD**, and **0.25**



22. **Repeat this for the leftHip**.

23. Select the **marchHare's rightKnee**, then select **one shots**, **procedures**, **turn**, **forward**, and **0.25**.

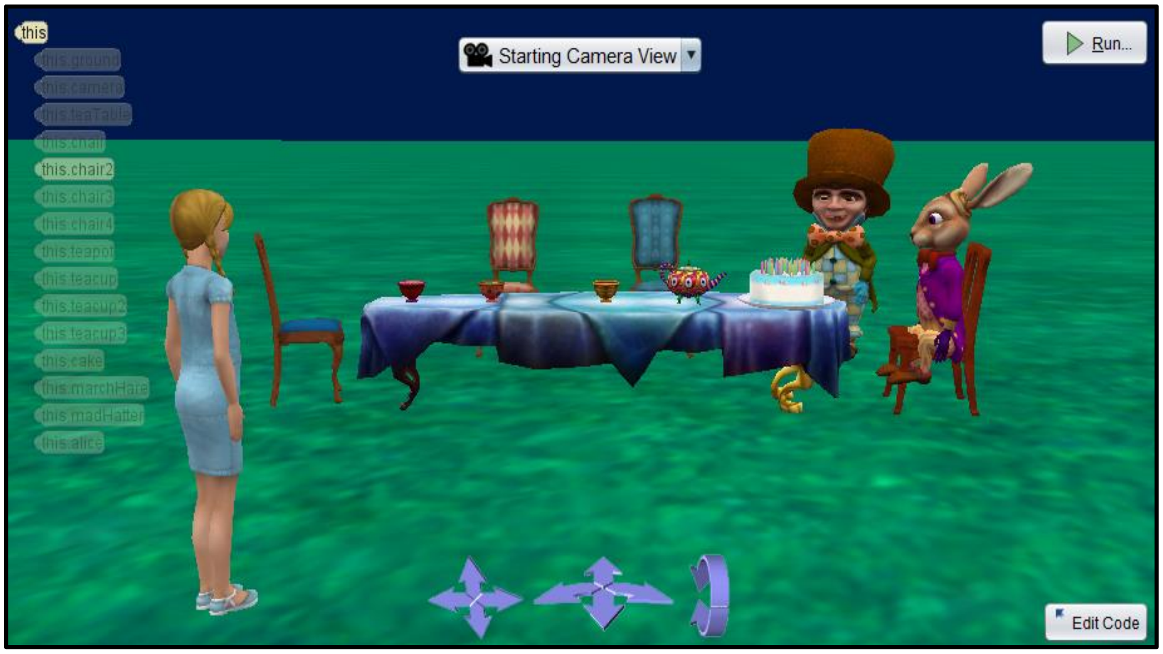
24. **Repeat this for the leftKnee**. You may need to move the entire marchHare back and up to get him onto the chair.



25. Now, let's **add the madHatter** to the scene. Place him next to the marchHare. It doesn't matter which side he is on. You may need to resize, rotate, and move him to get the scene to look the way you want.

26. Finally, we are going to add Alice to the scene. We will need to create Alice using the Child class in the biped classes. The Child class allows you to select male or female, the skin tone, the attire, the hair color, eye color, and shape of the person. **Create a girl** that looks like Alice and name her **alice**. Normally you would capitalize a name, but when we name objects, we don't capitalize the object names.

27. Place alice off to the side of the animation window looking at the tea party as shown below.



28. We are finished with the scene setup. If you want to add some wonderland trees or other objects to your scene, feel free.

29. We have not added any code to our projects. This chapter was all about working with scene setup. We will be adding code in chapter 1.

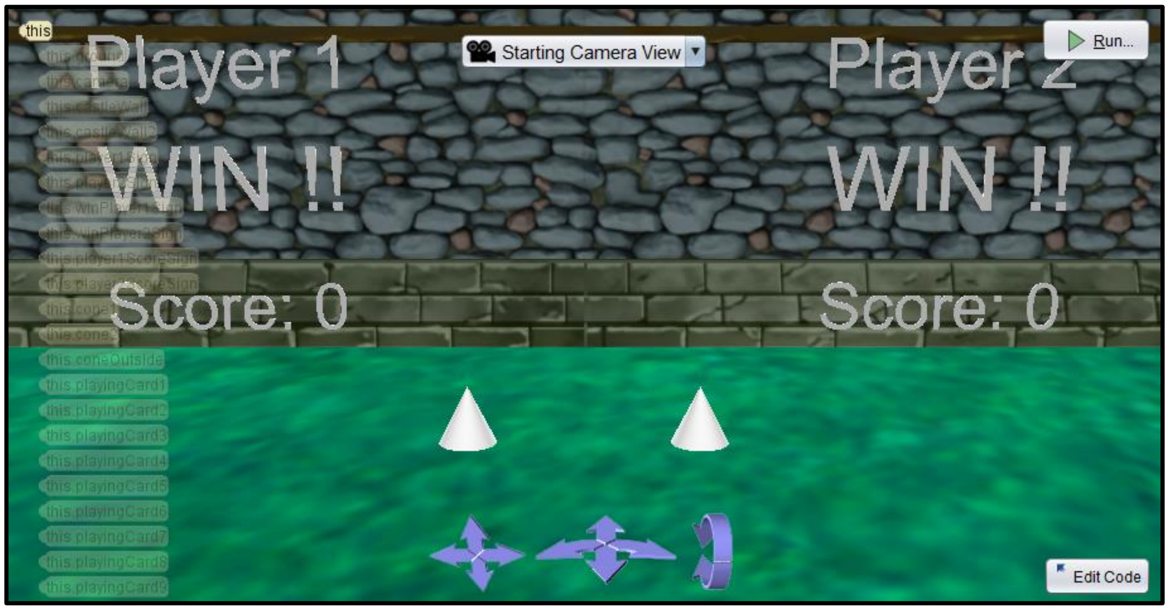
30. Save this program and exit Alice.

Exercise 4: Alice Card Game Scene Setup (ongoing exercise)

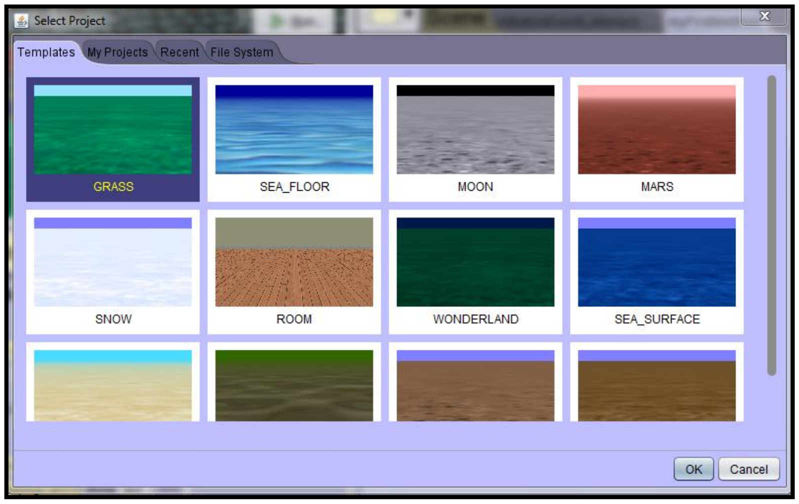
1. Open up Alice 3. You will need to find the installed Alice 3 folder and double click on the Alice 3 application file.

2. The first step in programming is understanding the problem. We would like to create a trimmed version of a card game. We will have 2 players that each get dealt a playing card. The player with the highest card wins. We will be gradually adding to this project throughout the text. In this project, we are going to set up the scene for the card game. We will need to add the playing cards to the scene and we will need cones to set the locations for our playing cards so that it looks as if the cards are being dealt. We will want to add text to the screen that displays each player, who won, and the score for each player. The castle wall will just act as a background for our game (it is not necessary).

3. Our goal for the scene setup is to have a castle background with the following characters: 10 Alice Playing Cards, **Player 1** text sign, **Player 2** text sign, 2 **WIN** signs (one for each player), 2 scores (one for each player). When we are finished it should look similar to the following:

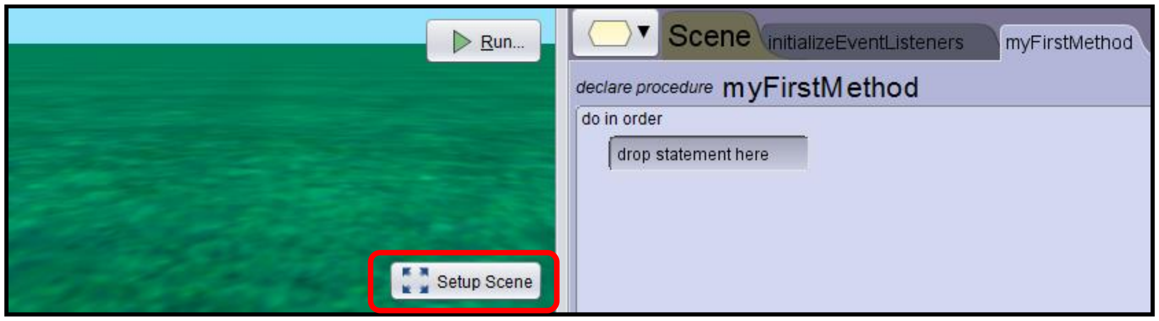


4. Select the **grass** template:

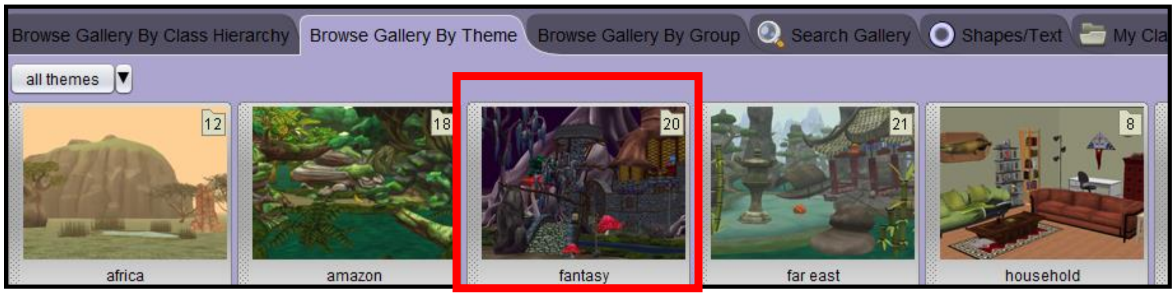


5. Select **File** from the menu, then **Save As**. Save this file as **CardGame**. Please get in the habit of capitalizing the first letter of every word in your filename and do not use spaces when naming your files. You should save your work often. You can click Save from the File menu from this point on. You should save this file in your **Chapter0Exercises** folder.

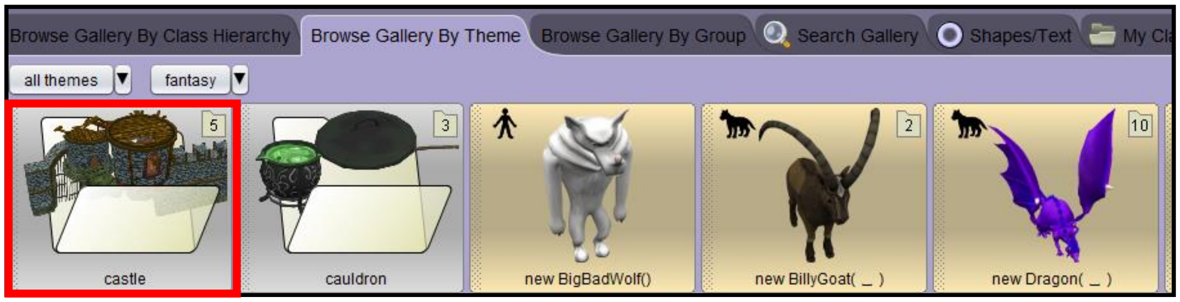
6. Click on **Setup Scene button.**



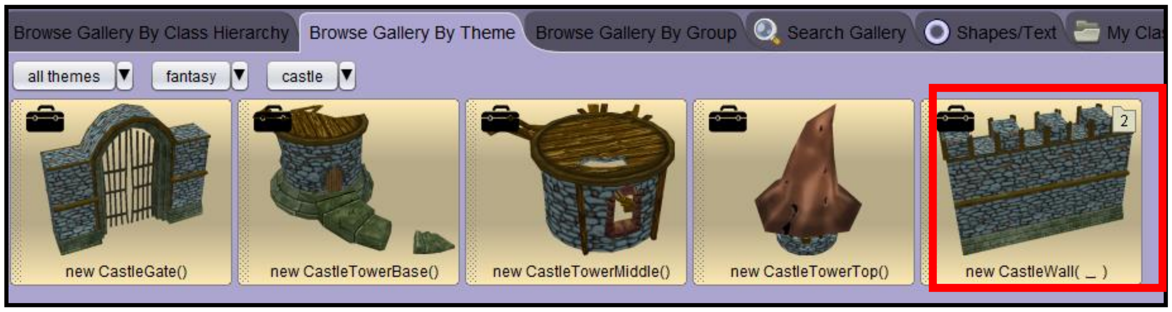
7. To add a castle wall to the background, click on the tab called **Browse Gallery By Theme.** Then click on **fantasy**.



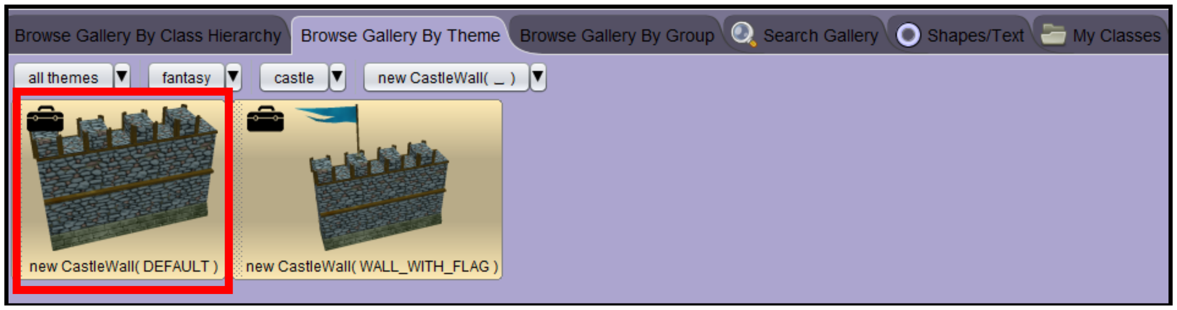
8. Next, choose **castle.**



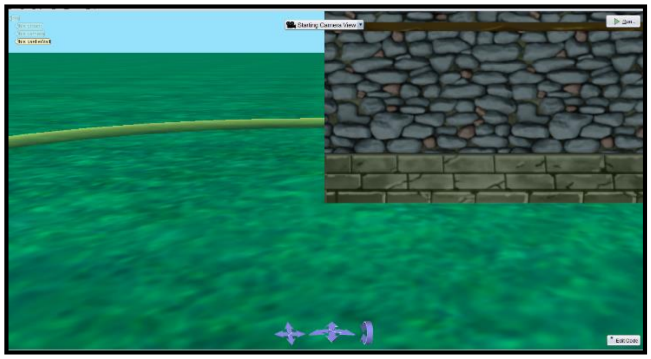
9. Then choose **new CastleWall** (last choice).



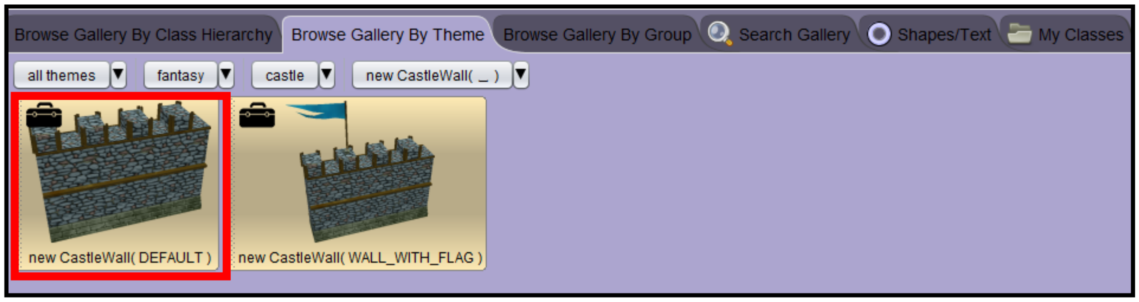
10. Then, choose the first choice for the castle walls.



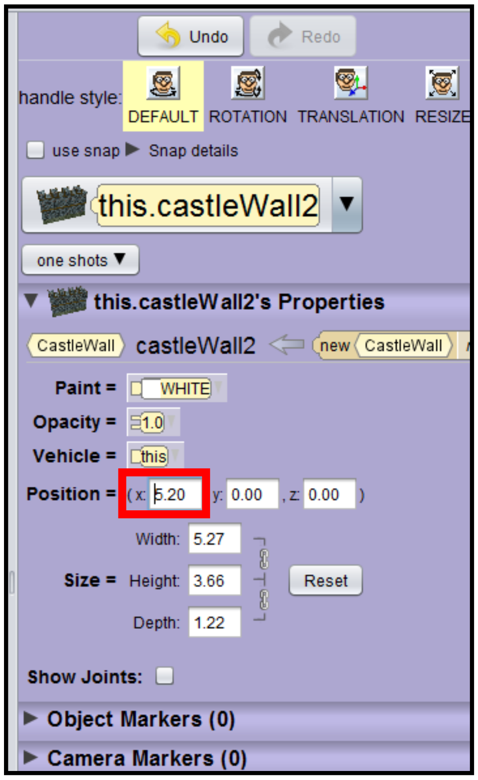
11. You can use the default name given as **castleWall** and click on OK. Your screen should look similar to the following:



12. Click on the castle wall choice at bottom of screen again so that you will have 2 castle walls. It will name this piece as **castleWall2** and that is fine. Click on OK. It will put this piece of the wall right over top of the last piece that was already on the screen.

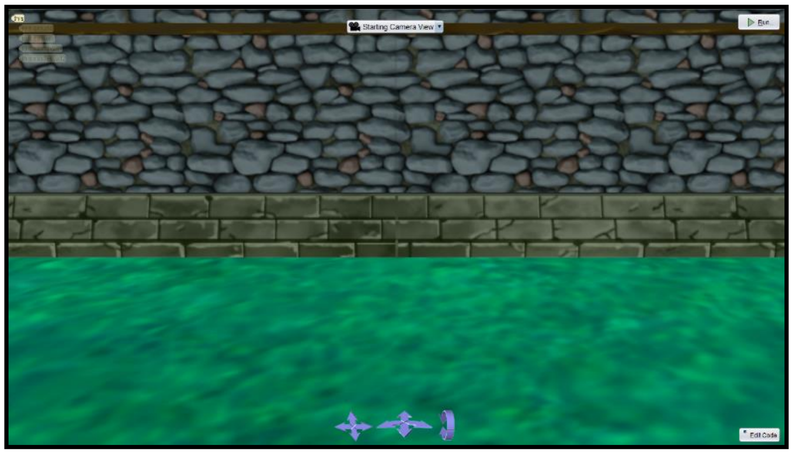


13. You can use the handles to pick this piece up and move it to the left until it looks like one continuous wall. Another way is to move it using the X Y Z axis positions listed on the far right of your scene setup screen. Change the **x** location to be **5.20** and press ENTER key and it will move it to the left.

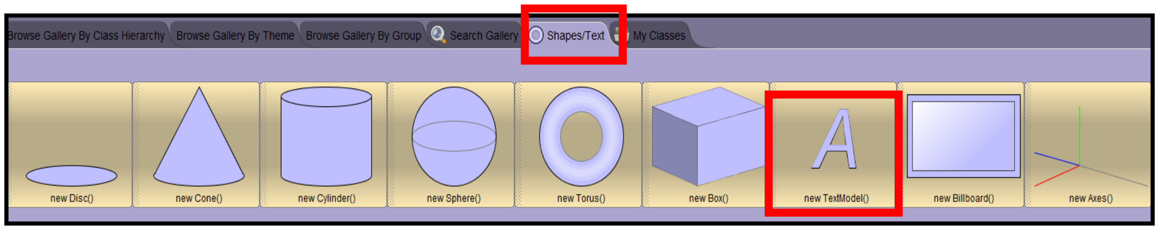


Paint = WHITE
Opacity = 1.0
Vehicle = this
Position = (x: 5.20 , y: 0.00 , z: 0.00)
Size = Width: 5.27 Height: 3.66 Depth: 1.22

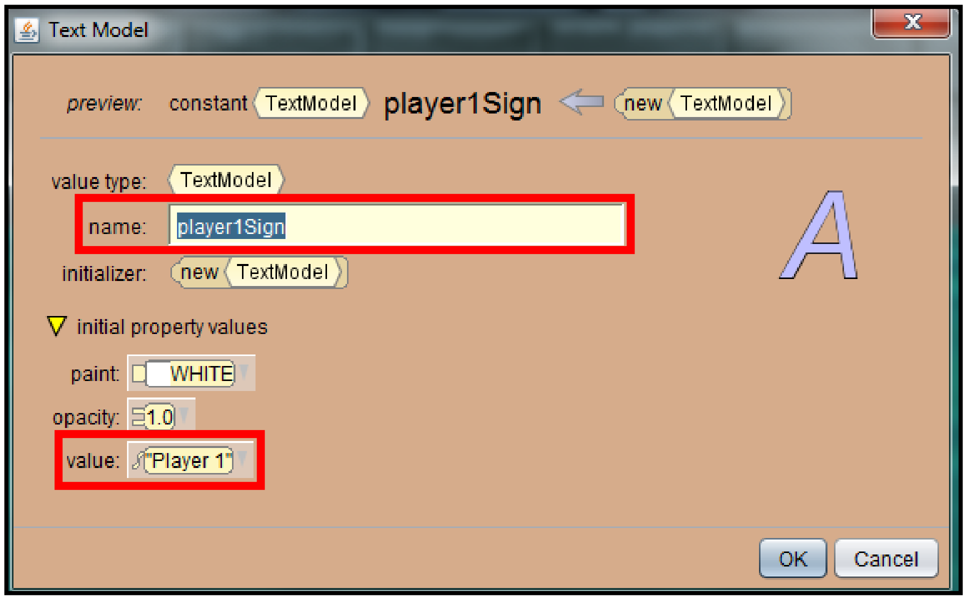
14. The screen should look similar to the following:



15. The background for the Card Game is now set up. Now, player signs are needed in the upper corners. These are text boxes. Choose the **Shapes/Text** tab and select **new TextModel** and **drag it onto the grass**.



16. Name the text model as **player1Sign**. By default, the color is white and the opacity is 1 (visible). Change the value by clicking in value and choosing custom textString and then typing in **Player 1** (with a space). Your screen should look as follows and then click **OK**.



preview: constant TextModel player1Sign ⇐ new TextModel()

value type: TextModel

name: player1Sign

initializer: new TextModel()

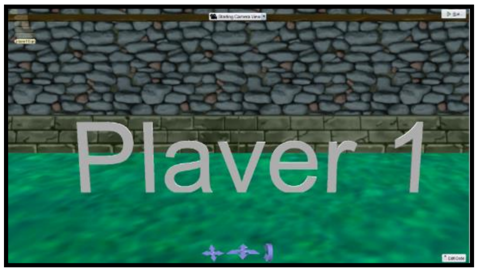
initial property values

paint: WHITE

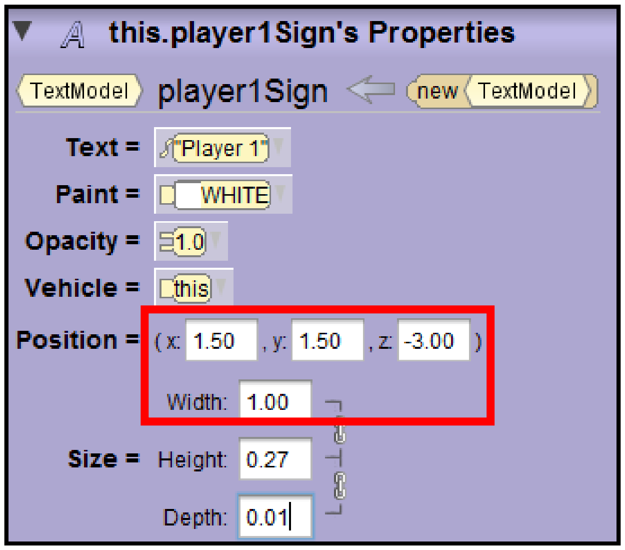
opacity: 1.0

value: "Player 1"

17. A text box will appear quite large in the grass as follows *(if you cannot see the text, do not worry it will be fixed in the next step)*:



18. To place this sign in upper left corner it will need to be resized and moved. One way is to move and resize the object by using arrows, etc. Another way is to use the right side of the screen to place it at a specific location by using the x, y, and z etc. Change the **width** to be **1** and press ENTER key. Change the **x, y, and z** to be as follows (make sure to press ENTER key after making each entry):

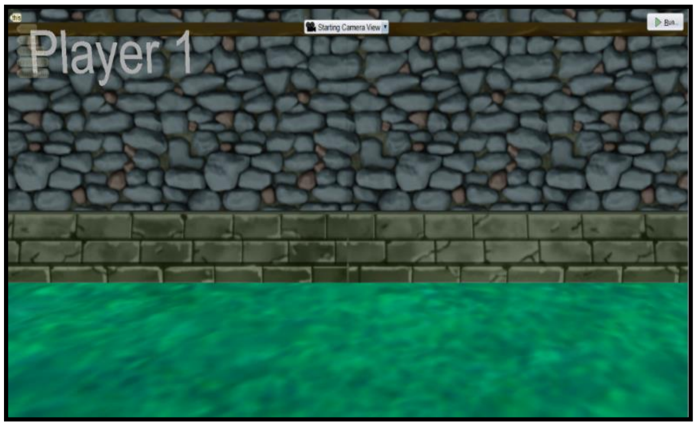


this.player1Sign's Properties

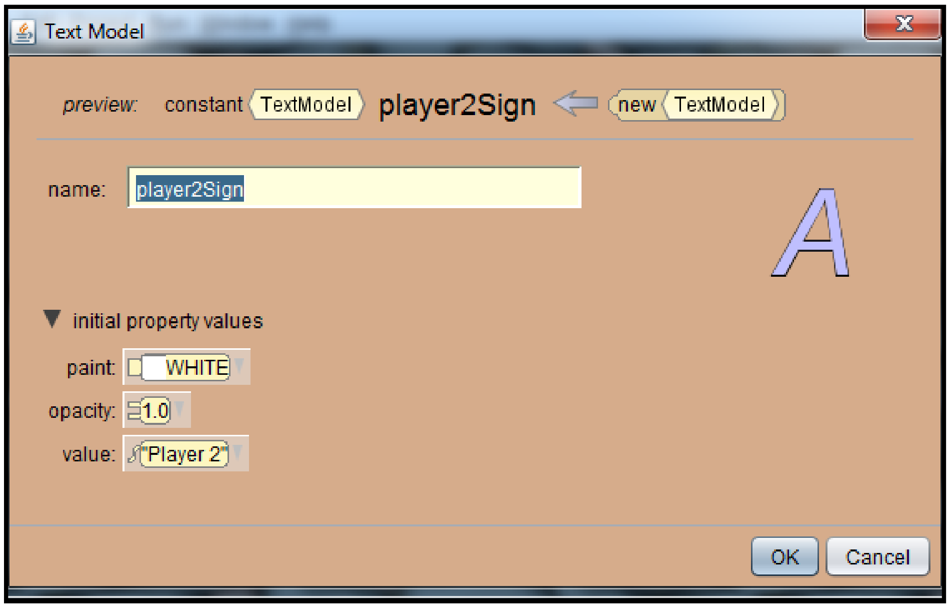
TextModel player1Sign ⇐ new TextModel

Text = "Player 1"
Paint = WHITE
Opacity = 1.0
Vehicle = this
Position = (x: 1.50 , y: 1.50 , z: -3.00)
Width: 1.00
Size = Height: 0.27
Depth: 0.01

19. The screen should appear as follows:



20. The Player 2 sign will be done the same way. Drag the **new TextModel** into the grass. Name it **player2Sign** and change the custom testString to be **Player 2**.



Text Model

preview: constant TextModel player2Sign ⟸ new TextModel

name: player2Sign

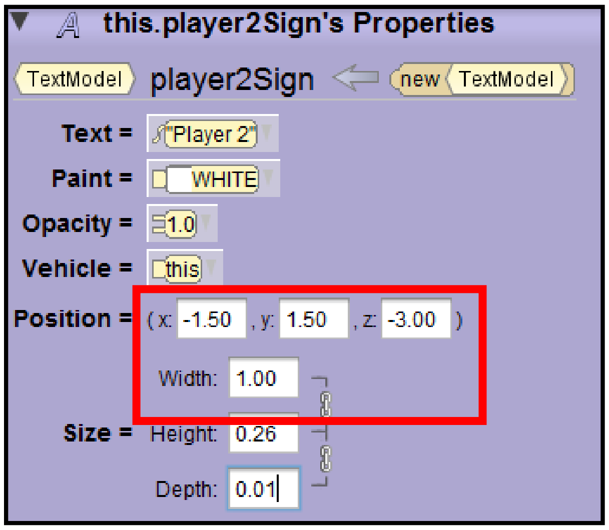
initial property values

paint: WHITE

opacity: 1.0

value: "Player 2"

21. The player2Sign comes in large. The width and location will need to be adjusted. Change the width and the x, y, and z to be as follows (make sure to press ENTER key after making each entry):



this.player2Sign's Properties

TextModel player2Sign ⟸ new TextModel

Text = "Player 2"

Paint = WHITE

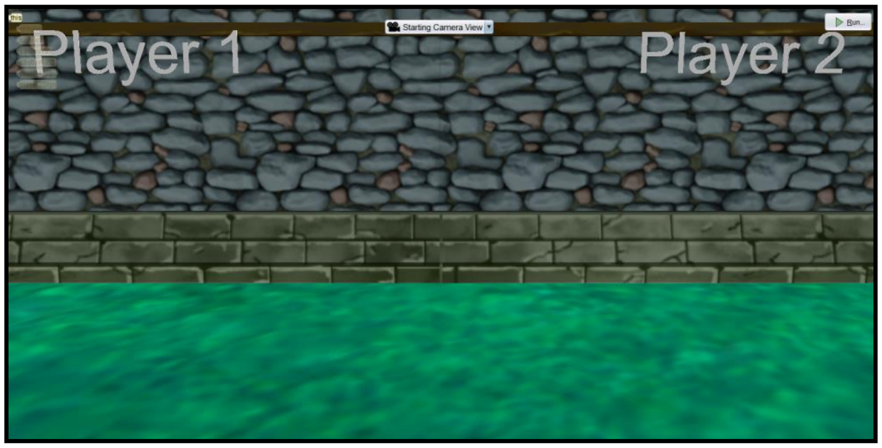
Opacity = 1.0

Vehicle = this

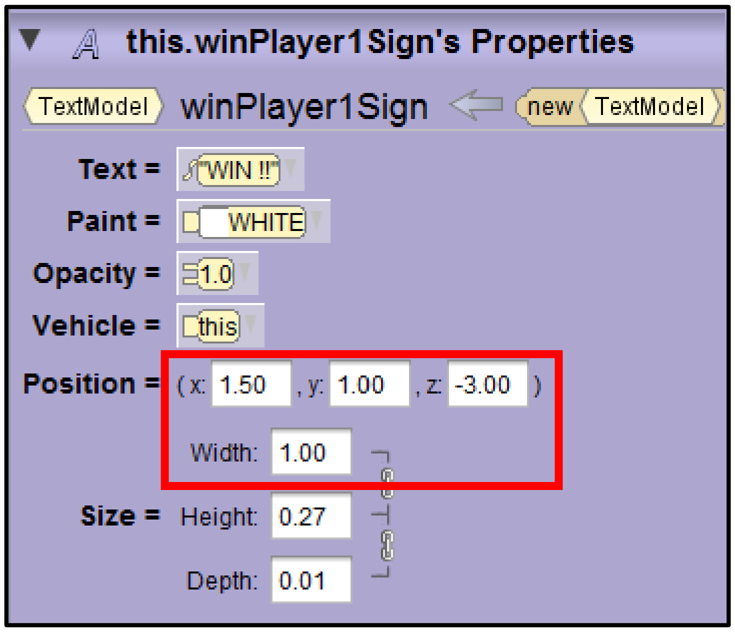
Position = (x: -1.50 , y: 1.50 , z: -3.00)

Size = Width: 1.00 Height: 0.26 Depth: 0.01

22. Your screen should look as follows:



23. The player 1 and 2 signs are headings. The WIN signs and scores will be displayed underneath of these. They will also be Text Models. The first WIN sign will appear directly under the Player 1 sign. Choose the **TextModel** again and drag it to the grass area. Fill in the box with a name of **winPlayer1Sign** and custom textString value as **WIN !!** Change the width and the positions as follows:



this.winPlayer1Sign's Properties

TextModel winPlayer1Sign ⇐ new TextModel

Text = "WIN !!"

Paint = WHITE

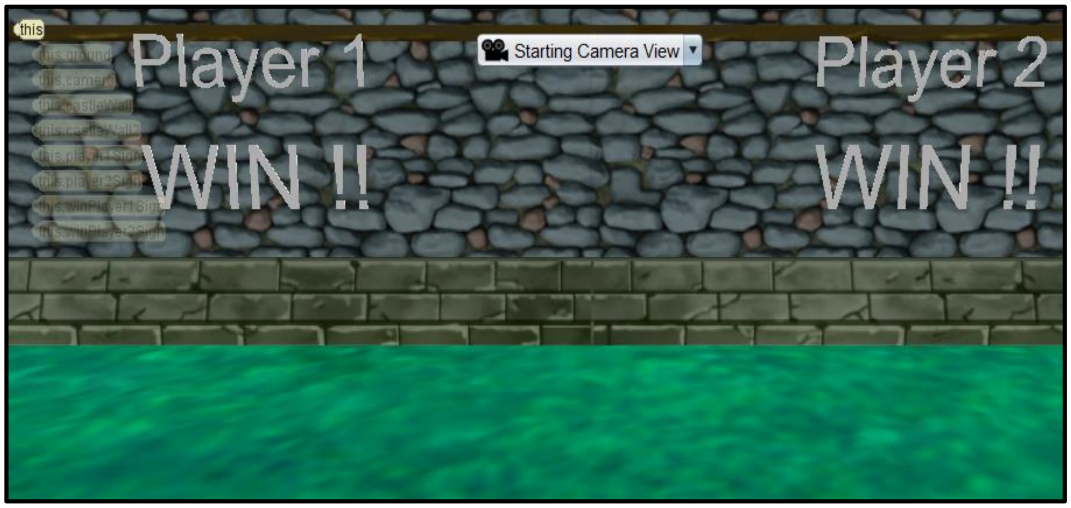
Opacity = 1.0

Vehicle = this

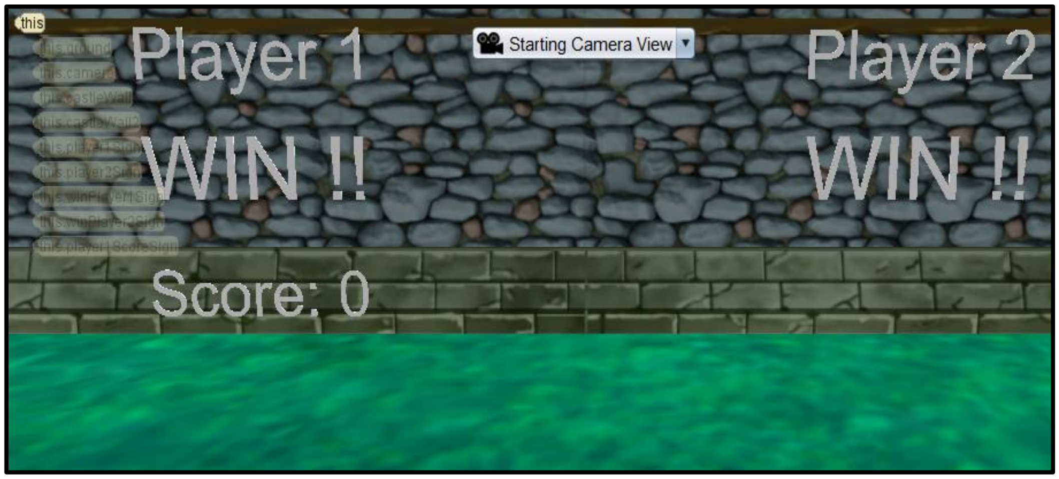
Position = (x: 1.50 , y: 1.00 , z: -3.00)

Size = Width: 1.00 Height: 0.27 Depth: 0.01

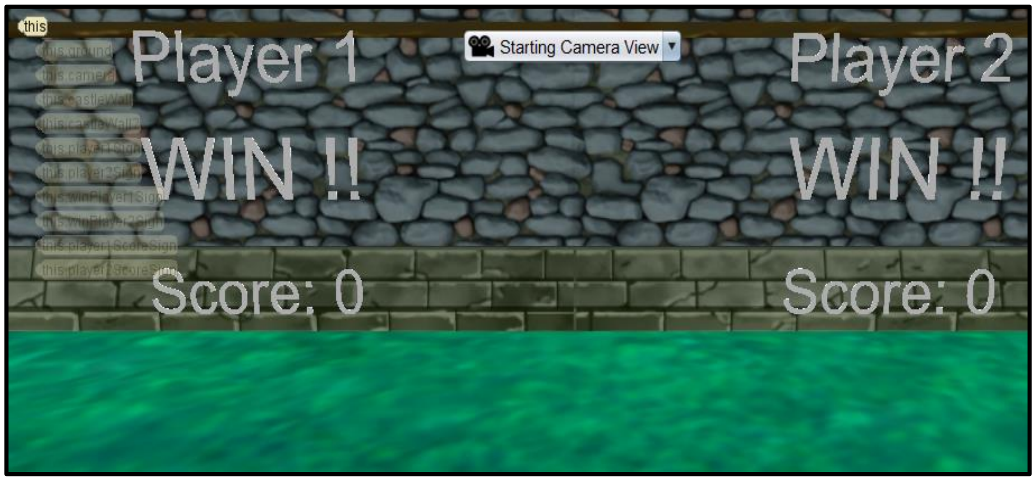
24. You will do the same thing for **winPlayer2Sign**. It will be named winPlayer2Sign and the textString value will be **WIN!!** Its width will be **1.0**. Its position will be x of **-1.5**, y of **1.0**, and z of **-3.0**. Make sure you press ENTER after each entry. If you have done everything correctly, your screen will look as follows:



25. Below these signs will be the SCORE signs. To set these up, drag another **TextModel** to the grass. Name it **player1ScoreSign**. The custom textString value will be **Score: 0**. The width is **1.0**. The x position is **1.5**, y position is **0.5**, and z position is **-3.0**. Your screen should look as follows:

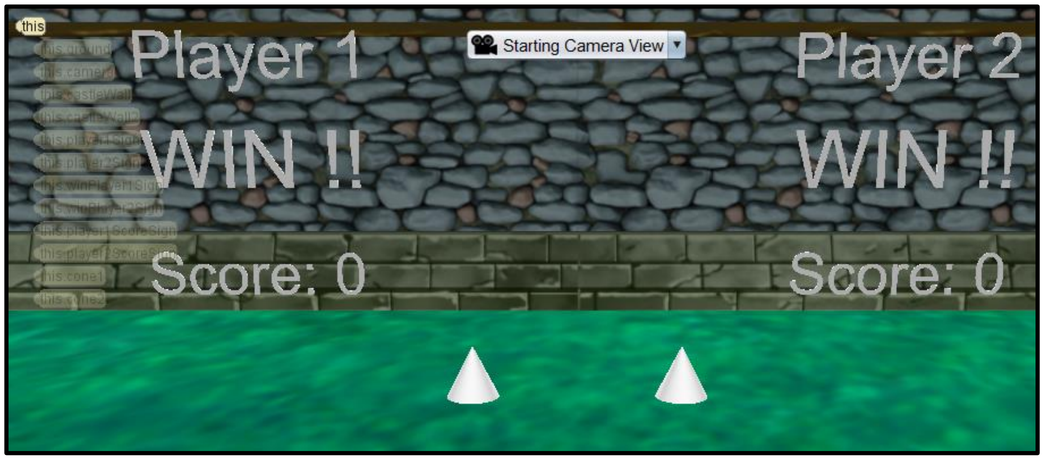


26. To do the second SCORE sign on right side, you will drag a **TextModel** to the grass again. Name the object as **player2ScoreSign** and make the custom textString as **Score: 0**. Change the width to be **1.0** and press ENTER key. Change the x position to be **-1.5**, y position to be **0.5**, and z position to be **-3.0**. Your screen should look as follows:

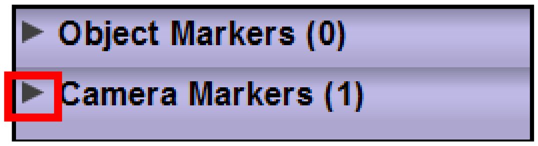


27. Two placement markers will be needed as markers for the playing cards. Cones will be used for these markers. Two small cones will be placed towards the middle of the scene and the cards will be dealt to these cone markers. From the Shape area, drag the cone to the grass. Give it the name of **cone1**. Change the width to be **0.25**. Change the x position to be **0.5**. Change the y position to be **0.0**. Change the z position to be **-3.0**. This could have also been done by using the arrows and lining up the positions by trial and error.

28. Now, place the second cone with a name of **cone2**, width of **0.25**, x position of **-0.5**, y position of **0.0**, and z position of **-3.0**. The screen should look as follows:

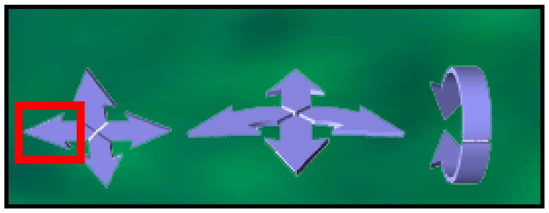


29. Another marker needs to be placed far off the scene. Since camera movement will be involved, a marker will be placed to remember this camera position so we can return to this camera angle later. On the right side of screen by the sizes and locations is a section called **Camera Markers**.



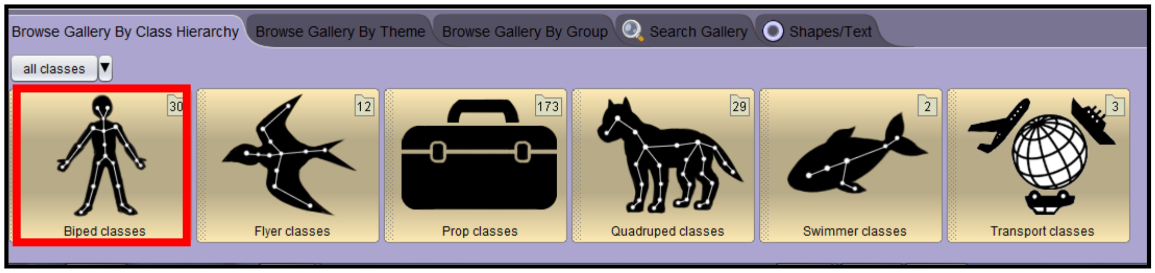
Click on the triangle (expand) to open the Camera Markers part of the screen. Choose the **Add Camera Marker…** button. Name this camera marker as **originalCamera** and make it RED. Now, we will be able to return to this view any time we want. (It is good idea to have an original camera marker for any scene you develop.)

30. Click on the first camera arrow at bottom of scene.

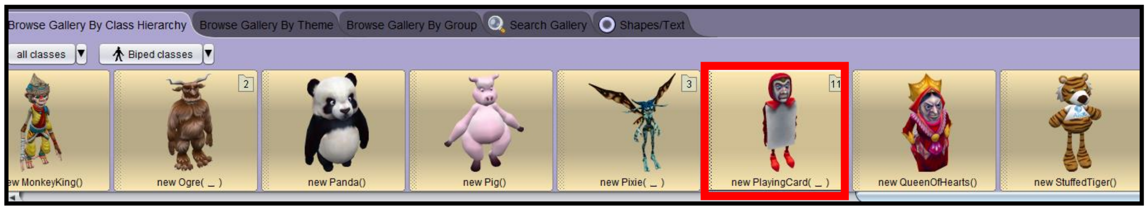


Choose the LEFT arrow. Your screen should scroll to the left. The goal is to scroll far enough to the left to place a marker to the left of the wall. Create a cone named **coneOutside**. Make the length of it **0.25**. Make the x position as **8.0**, y position as **0.0**, and z position as **-3.0**. Each of the playing cards will be placed at this same position. This way each of the cards will be at the exact same location and they will be off the original scene.

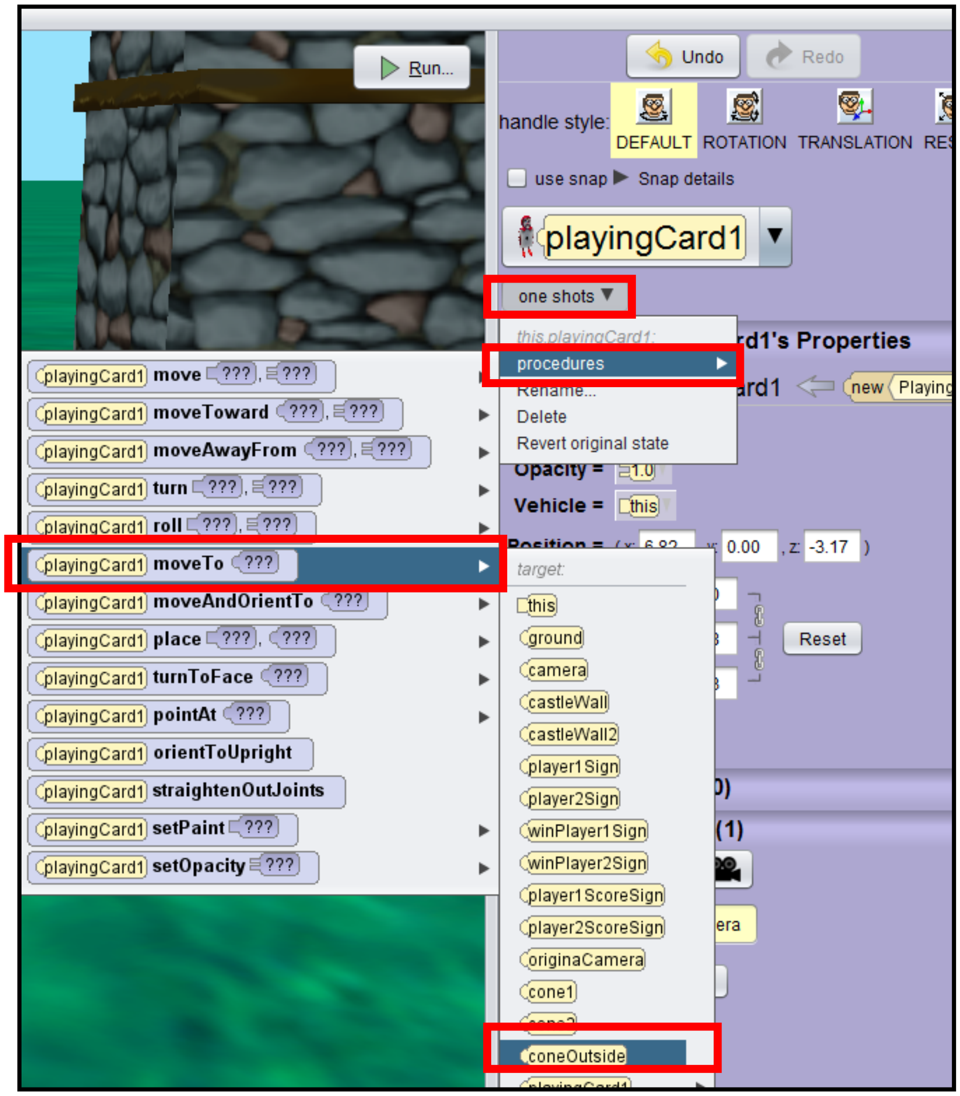
31. **To place the first card, we will click on the tab that is Browse** Gallery by Class Hierarchy and choose the **Biped Classes**



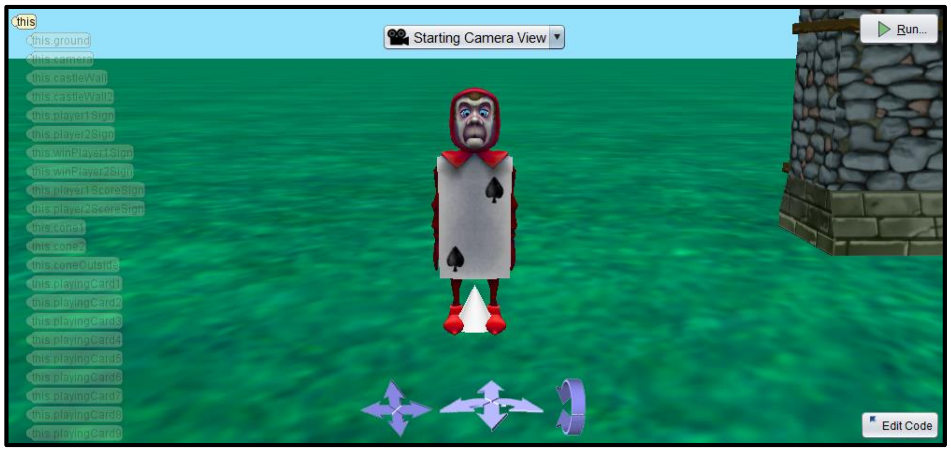
32. Find **new PlayingCard**.



33. Scroll right and choose new **PlayingCard (ONE 1)** and drag it to the screen. Name this card **playingCard1**. Change the width to be 0.5. Click on the **one shots** drop down (or right click on the object). Choose **procedures**, then **moveTo**, and then choose **coneOutside**.

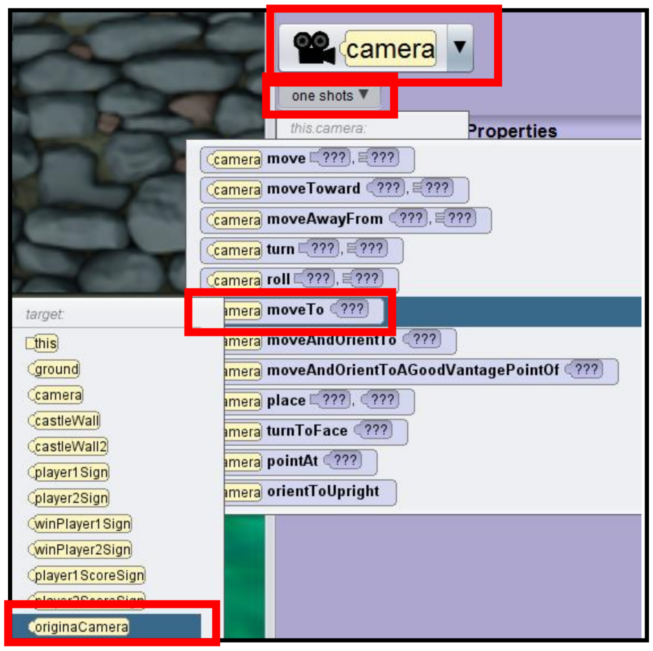


34. The first playing card should now be placed exactly on the coneOutside. Now, nine more playing cards will be placed in this exact same location and the exact same size. To place the second card, scroll to the right of the playing cards and find **newPlayingCard (TWO2)**. Drag it to the grass. It will automatically name it **playingCard2**. Change the width to be **0.5** and press ENTER key. Use the one shots to place playingCard2 on the coneOutside as you did in the previous step. You will continue to place all ten cards on the screen in the grass, size each one as a width of **0.5** and the use one shots to **move each one to the coneOutside**. When all are done, your screen should look as follows:

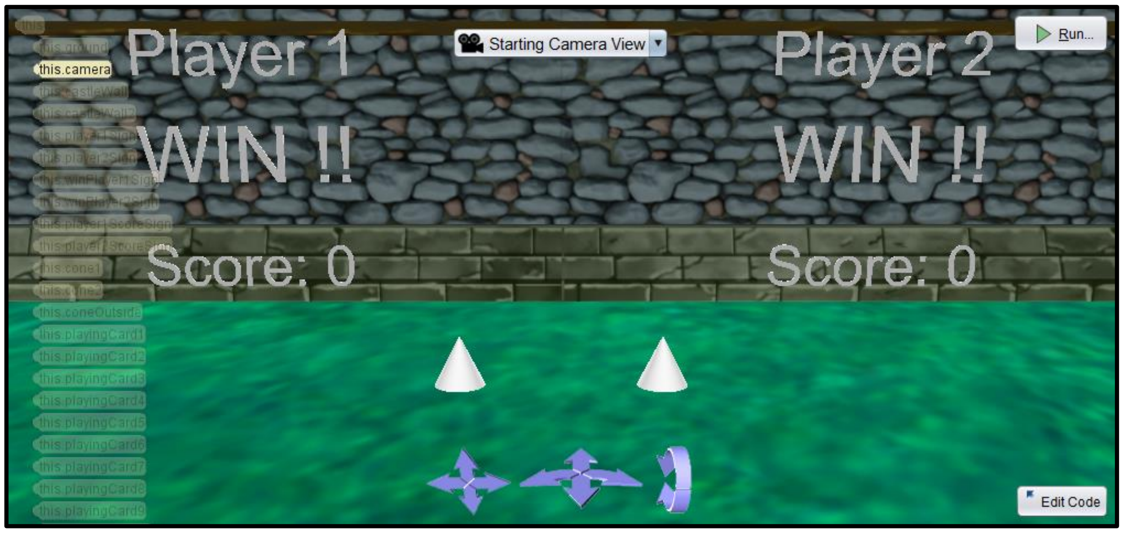


If done correctly, the object tree on left should have playingCard1 to playingCard10 and all of the cards should be sitting on top of each other and on top of our coneOutside.

35. To return to the original camera view, change object to **camera** and then choose **one shots**, **procedures**, **moveTo**, and **orignalCamera**.



36. In later chapters, you will have these cards come on to the screen, determine which player is the winner, have the WIN sign flash, have the score updated, etc. The scene is now complete and should look as follows:



37. Make sure to save this final version as **CardGame** in the Chapter0Exercises folder.

Summary

- Alice is an innovative 3D programming environment that makes it easy to create an animation or interactive game. The team named the system “Alice” in honor of Lewis Carroll who wrote Alice’s Adventures in Wonderland.
- An Alice virtual world begins with a template for an initial scene. These templates can be grass, water, snow, etc.
- An Alice 3D model is like a blueprint that tells Alice how to create a new object in the scene. The 3D model provides instructions on how to draw the object, what color it should be, what parts it should have, its size (height, width, and depth), and many other details.
- When you choose to place an Alice object into your world, Alice will create an object (instance) of that class in your world and ask you to name that object.
- When naming an object (instance) of a class, you should begin the name with a lowercase letter. If the name will have multiple words in it, each successive word with then begin with a capital letter. An example would be ***myLittleSnowman***.
- Objects from the galleries are added to the scene via the SCENE EDITOR (a click on Setup Scene button).
- Objects in an Alice world are three dimensional. Each object has width, height, and depth.
- There are six possible directions in which an object may move – forward, backward, up, down, left and right.
- Each object in Alice has a unique “center.” An object’s center is used for measuring distance to another object and for determining its position in the world.

Review Questions

1. Alice was named in honor of Lewis Carroll.
   a. True
   b. False

2. An Alice 3D model is like a blueprint that tells Alice how to create a new object in the scene.
   a. True
   b. False

3. Once an object is placed into a scene, it can't be manipulated by moving, rotating, etc.
   a. True
   b. False

4. If you were to name an object (instance) of an Airplane class, which of the following names would be proper?
   a. MyAirplane
   b. my airplane
   c. myairplane
   d. myAirplane

5. You can have more than one object of the same class in the same world?
   a. True
   b. False

Solutions: *1) a   2) a   3) b   4) d   5) a*

Assignments

0-1 **Cola Commercial:** Create a scene in Alice that could be used for a cola commercial. You must have at least 7 appropriately placed objects in your scene. Name this project ColaCommercial.

0-2 **Greeting Card**: Create a scene in Alice that could be used for a greeting card. You must have at least 7 appropriately placed objects in your scene. Name this project GreetingCard.

0-3 **Animation**: Create a scene in Alice that could be used for a short animation. You must have at least 7 appropriately placed objects in your scene. Name this project as Animation.

0-4 **Card Game Adjusted**: Adjust exercise 4 in this chapter to have at least 7 more appropriate scenery items. Name the project as MyCardGame**.**

Chapter 1

Coding Introduction

Objectives

- ☑ Explain the difference between high and low level programming languages
- ☑ Describe the history of how the Java programming language was started
- ☑ Briefly describe the following:
  - Object Oriented Programming
  - Platform-Independence
  - Garbage Collection
  - Java Development Kit
- ☑ Explain the difference between applets, applications, and servlets
- ☑ Explain the difference between Java and JavaScript
- ☑ Compile and execute a Java program
- ☑ Debug errors
- ☑ Identify and fix compiler errors

Introduction to Programming

A computer program is a way to tell a computer what to do. When you want a computer to perform a task, you must give it line-by-line instructions on how to accomplish that task. These line-by-line instructions are called a **computer program**.

The computer stores information based on electronic signals, referred to as **binary**. A **bit** (binary digit**)**, the smallest unit of information storage, is represented by either an on (1) or off (0) signal inside the computer. One **byte** (a character such as the letter "A" on the keyboard) uses eight bits.

There are many different computer programming languages available and the choice of what programming language to use will depend upon the task for the computer to accomplish. A programming language that is written at the very low technical circuitry level of the computer is called a **low-level programming language**. Some examples of low-level programming languages are machine language and assembler language. Machine language is composed of binary 1's and 0's and is not intended for humans to read. Machine language varies from computer to computer. The machine language for a PC is entirely different from machine language for Mac. A computer only understands programs (without any conversion) written in its machine language (binary).

High-level programming languages allow programmers to write programs using English terms. Computers do not understand high-level languages directly so this means that computer programs written in a high-level language must be converted to machine language by an interpreter or compiler. Some high-level computer programming languages available are: C++, Visual Basic.NET, C#, and Java. Each of these programming languages is best-suited to a certain type of computer or problem such as mainframes, business, games and/or science.

Computer languages each have their own **syntax**, or rules of the language. For instance, in a high-level programming language the verb to display information might be "write", "print", "show", etc. In a low-level programming language the verb to display information might be a code of "101011" in binary. ***Java is a high-level programming language with a specific vocabulary and specific rules for using that vocabulary.***

What is Java?

History of Java

In 1990, **James Gosling** was given the task of creating programs to control consumer electronics (TVs, VCRs, toasters, etc.). Gosling and his team at **Sun Microsystems** started designing their software using C++. The team found that C++ was not suitable for the projects they had in mind. They ran into trouble with complicated aspects of C++ such as multiple inheritances of classes and with program bugs such as memory leaks. So, Gosling created a simplified computer language that would avoid all the problems he had with C++. Thus, **a new programming language named Oak** (after a tree outside his window) was born.

Oak was first used in something called the Green project, which was a control system for use in the home using a hand-held computer called Star Seven. Oak was then used in another project which involved video-on-demand. Neither project ever made it to the public eye, but Oak gained some recognition. Sun discovered that the name Oak was already copyrighted. After going out for coffee one day, they named their new powerful language **Java**.

In 1993, the Java team realized that the Java language they had developed would be perfect for web page programming. The team came up with the concept of web applets, small programs that could be included in web pages, and created a complete web browser called HotJava (originally called Webrunner) that demonstrated the language's power.

In the second quarter of 1995, Sun Microsystems officially announced Java. The "new" language was quickly embraced as a powerful tool for developing Internet applications. Netscape Communications added support for Java to its Netscape Navigator 2.0. Java became an instant "hit" and also made the Netscape browser very popular. Other Internet software developers such as Microsoft eventually followed suit and reluctantly included Java in their browsers. These browsers were called "Java-enabled". Java-enabled meant that the browser could download and play Java classes (applets) on the user’s system. (Applets appear in a web page much the same way as images do, but unlike images, applets can be dynamic and interactive.)

Java Capabilities

- **Java is easier than C++**. Although Java looks similar to C and C++, most of the complex parts such as pointers, multiple inheritance, and memory management have been excluded from Java.

- **Java is an Object Oriented Programming (OOP) language,** which allows you to create flexible, modular programs and reuse code. OOP is based on the theory that everything in the world can be modeled as an object. An object has attributes (data) and behavior (methods).

- **Java is platform-independent.** Platform-independence is a program’s capability of moving easily from one computer system to another. **Java slogan is "You can write once and run anywhere."** If you write a game using the Java programming language, theoretically, you should be able to run that game on a PC, Linux, or Mac.

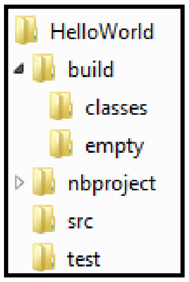
- **Java supports the Internet** by enabling people to write interactive programs for the Internet. Java applets can easily be invoked from web browsers to provide valuable and spectacular web pages.

- **Java is general purpose.** Although used mainly for writing internet applications, Java is a truly general-purpose language. Almost anything that most other computer programming languages such as C++ or Visual Basic can do, Java can also do. Java programs can be applets for the Internet or standalone applications for local PCs.
    - **Applets** appear in a web page much in the same way as images do, but unlike images, applets are dynamic and interactive. Applets can be used to create animations, games, ecommerce, etc.
    - **Applications** are more general programs written in the Java language. Applications don't need a browser. The Java language can be used to create programs, like those made in other computer languages.
    - **Servlets** are programs that respond to requests from clients.
- **Java is secure**. Since the Java program is isolated from the native operating system of a computer, the Java program is insulated from the particular hardware on which it is run. Because of this insulation, the Java Virtual Machine provides security against intruders getting at your computer's hardware through the operating system.
- **Java programs can contain multiple threads** of execution, which enables programs to handle several tasks simultaneously. For example, a multi-threaded program can render an animation on the screen in one thread while continuing to accept keyboard input from the user in the main thread. All applications have at least one thread.
- **Java has multimedia capabilities** of graphics, images, animations, audio and videos. It also runs on networks.
- **Java programs do their own garbage collection**, which means that programs are not required to delete objects that they allocate to memory. This relieves programmers of virtually all memory-management problems.
- **Java programs are reliable and robust**. When a serious error is discovered, Java programs create an **exception**. This exception can be captured and managed by the program and then terminated gracefully.
- **Java vs. JavaScript**. The Java language was developed by Sun MicroSystems and is full programming language that can be used in applications or as applets on the Internet. JavaScript was developed by Netscape as a scripting language to be used only in HTML web pages.

Programming Process

Develop an algorithm: Think about the problem before coding. Create a flowchart, storyboard, or pseudo code to represent a solution to the problem.

Create Project: Create a new project in NetBeans. The NetBeans environment is known as our **IDE** (Integrated Development Environment). There are many IDEs that can be downloaded free of charge, but NetBeans provides many features that will be helpful to us for this course. In NetBeans, when you create a project, it creates a folder structure. The following is an example of a folder structure for a HelloWorld project created in NetBeans.

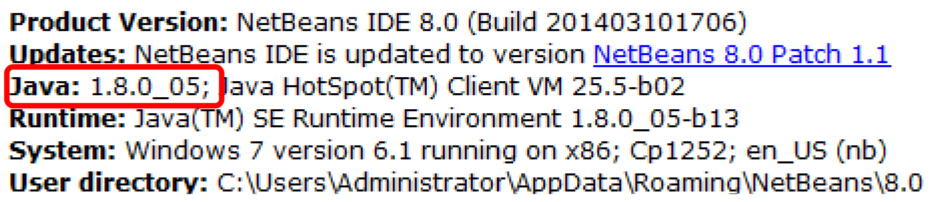


Code: Type the Java code. As you type, NetBeans checks your program for syntax errors. Red lines indicate errors in your code. The Java code (HelloWorld.java) for the HelloWorld project will be in the **src** folder in the folder structure above.

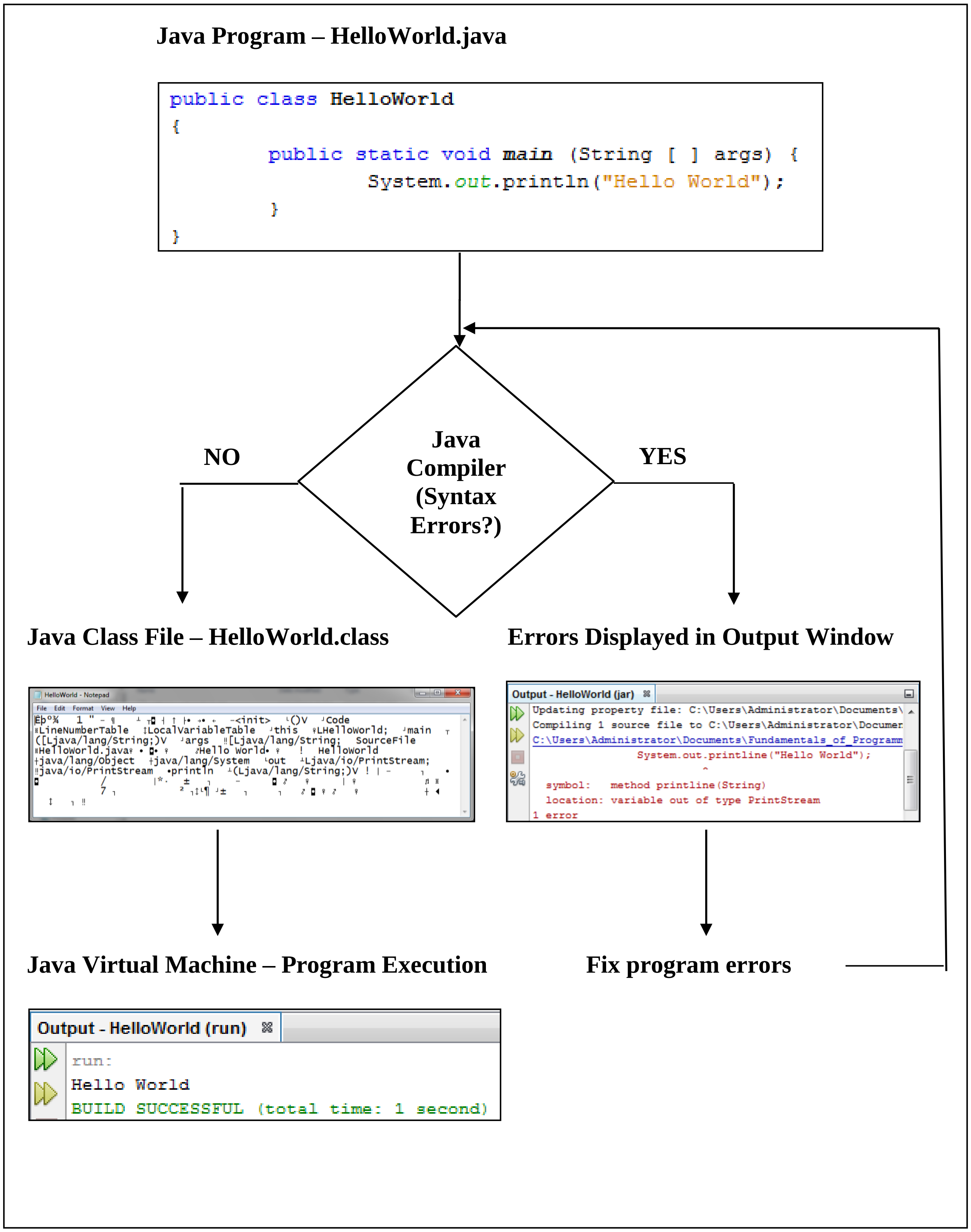
Compile: When you are finished typing the program, you will need to do a final compile of the program (also known as building). The Java compiler checks your code for errors. If it compiles with no syntax errors, it creates a **class file** (bytecode) that will be capable of running on different operating systems. Bytecode are a set of instructions that look a lot like machine code, but are not specific to any one processor. Compiling the HelloWorld.java creates the HelloWorld.class file located in the **build** folder inside the **classes** subfolder in the folder structure above.

Run: These bytecode are then fed to a **JVM** (Java Virtual Machine) where they are interpreted and executed.

The **JDK** (Java Development Kit) includes the Java library (code), JVM, as well as the Java compiler. The version of NetBeans that you installed included the JDK. Oracle owns Java and it is constantly releasing new versions of the JDK. It is good to know what JDK you are using so that you know what Java code is available to you. We will talk more about the Java library and JDK in a later chapter. You can check to see what version of the JDK that you have by clicking **Help** from the menu and then **About** in the NetBeans environment. The JDK version shown below is 1.8. You do not need to put the update number which is the number after the underscore.



Product Version: NetBeans IDE 8.0 (Build 201403101706)
Updates: NetBeans IDE is updated to version NetBeans 8.0 Patch 1.1
Java: 1.8.0_05; Java HotSpot(TM) Client VM 25.5-b02
Runtime: Java(TM) SE Runtime Environment 1.8.0_05-b13
System: Windows 7 version 6.1 running on x86; Cp1252; en_US (nb)
User directory: C:\Users\Administrator\AppData\Roaming\NetBeans\8.0


Java Program – HelloWorld.java
Java Compiler (Syntax Errors?)
NO
YES
Java Class File – HelloWorld.class
Errors Displayed in Output Window
Java Virtual Machine – Program Execution
Fix program errors
HelloWorld - Notepad
File Edit Format View Help
Output - HelloWorld (jar)
Output - HelloWorld (run)

```java
public class HelloWorld
{
          public static void main (String [ ] args) {
                    System.out.println("Hello World");
          }
}
```

```
Êþº¾   1 "  -<init>  ()V  Code
LineNumberTable  LocalVariableTable  this  LHelloWorld;  main
([Ljava/lang/String;)V  args  [Ljava/lang/String;  SourceFile
HelloWorld.java      Hello World    !  HelloWorld
java/lang/Object  java/lang/System  out  Ljava/io/PrintStream;
java/io/PrintStream  println  (Ljava/lang/String;)V !
```

```
Updating property file: C:\Users\Administrator\Documents\
Compiling 1 source file to C:\Users\Administrator\Documen
C:\Users\Administrator\Documents\Fundamentals_of_Programm
                    System.out.println("Hello World");
                                        ^
  symbol:   method printline(String)
  location: variable out of type PrintStream
1 error
```

```
run:
Hello World
BUILD SUCCESSFUL (total time: 1 second)
```

Documentation

Comments are used to document code so that other people reading our code can understand our logic. Comments are useful for adding extra information to our programs that we don't necessarily want to show up in the output of our program such as: author, date, JDK used, program description, etc. Also, it is a good idea to comment your programs extensively when you are just starting out so that you have well-documented examples.

A **single line comment** is represented by two forward slashes. This comment will continue until the end of the line. This type of comment can be placed on a line by itself or it can be placed on the end of a line of code to describe the code. The following are examples of a single line comments.

//Single Line Comment

System.out.println("Hello World"); **//Prints "Hello World" to the output window**

A **multi-line comment** is represented by a forward slash followed by an asterisk and an asterisk followed by a forward slash to end the multi-line comment. The following is an example of a multi-line comment.

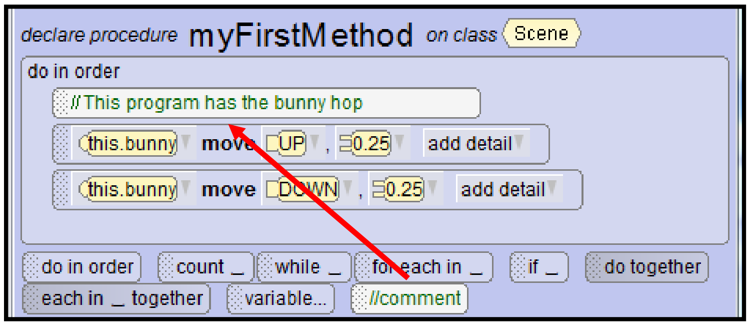
```
 /* Multi-Line
    Comment */
```

You can even be creative and separate your multi-line comments from your code by adding asterisks after the first forward slash and before the last forward slash.

```
/**
 * Multi-Line Comment
 * Typical Java Documentation
 */
```

Note: Java documentation will be explained in a later chapter.

Alice Comments



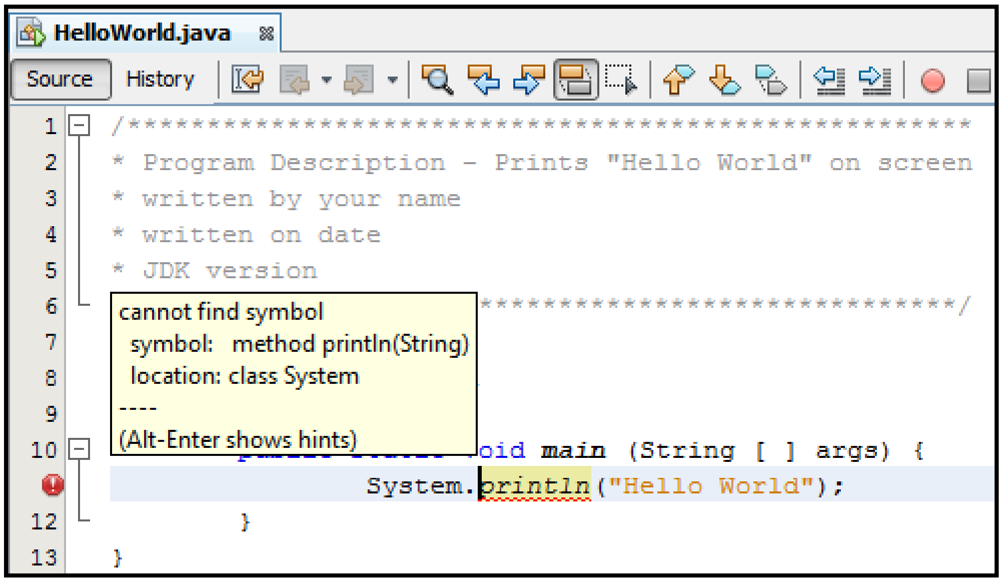
```
declare procedure myFirstMethod on class Scene
do in order
    //This program has the bunny hop
    (this.bunny) move UP , 0.25  add detail
    (this.bunny) move DOWN , 0.25  add detail
```

Program Errors

There are 3 different types of programming errors: compiler, run-time, and logic errors.

- **Syntax errors** are caused when the user writes code that is not understood by the compiler. A syntax error can be caused by incorrect capitalization or spelling mistakes. The compiler informs the user of a syntax error by displaying an error message. Typing “Public Class” instead of “public class” would result in a syntax error. NetBeans checks for errors as you type. If you see a red exclamation point before a line of code, you can hover over it with your mouse to see the error.

  This line of code should have been: **System.out.println("Hello World");**



```
/*****************************************************
* Program Description - Prints "Hello World" on screen
* written by your name
* written on date
* JDK version
*****************************/

        public static void main (String [ ] args) {
                System.println("Hello World");
        }
}
```

```
cannot find symbol
  symbol:   method println(String)
  location: class System
----
(Alt-Enter shows hints)
```

- **Run-time errors** are caused by invalid data. Run-time errors do not affect the compilation of your program thus the program will compile and execute, but it may crash or hang after execution. If you try to divide 12 by 0 you would get a run-time error because you cannot divide by 0.

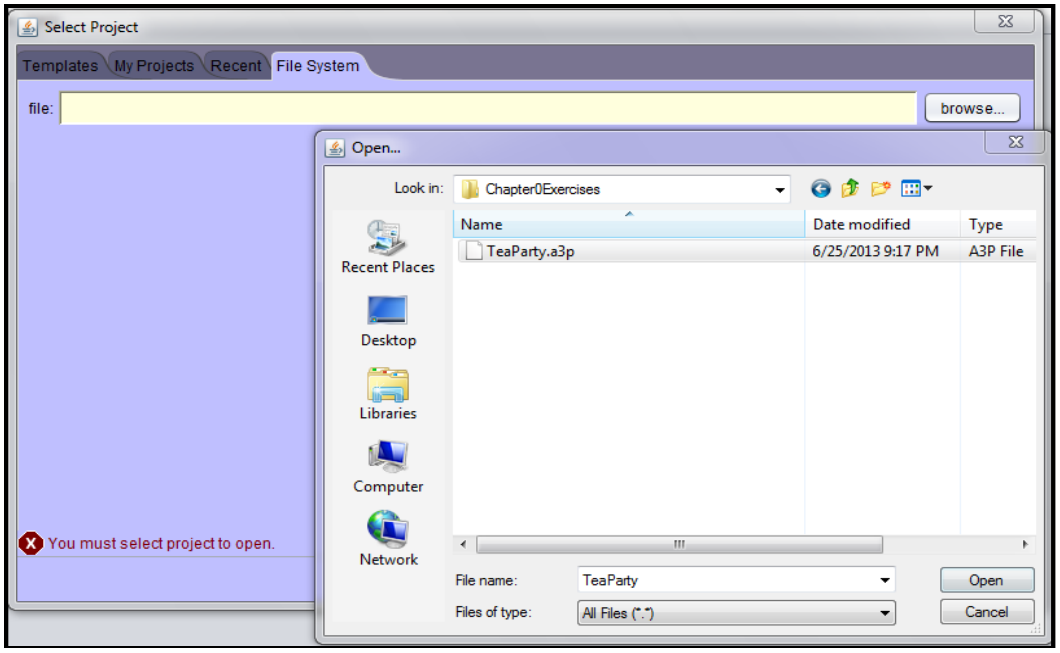
- **Logic errors** (also known as human error) are caused by mistakes that do not defy the rules of the language and do not crash or hang the program, but instead yield incorrect results. The user may not understand the problem that the program is trying to solve and therefore uses the wrong equation, wrong strategy, etc. An example of a logic error would be moving left instead of right.

Hands-on Exercises

Exercise 1: Alice in Wonderland Tea Party Coding (ongoing exercise)

1. Open up Alice 3.

2. Open the **TeaParty** file that was created in the **Chapter0Exercises** folder. Click on the **File System** tab, then choose **browse…** and locate your file. Select the file and click the **Open** button and then the **OK** button.



Name	Date modified	Type
TeaParty.a3p	6/25/2013 9:17 PM	A3P File

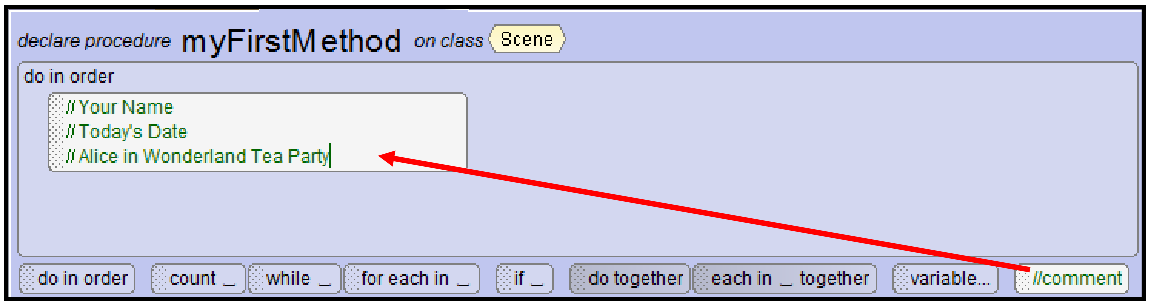
You must select project to open.

3. Before writing the code for our animation, we should first create a storyboard of what we wish to accomplish.

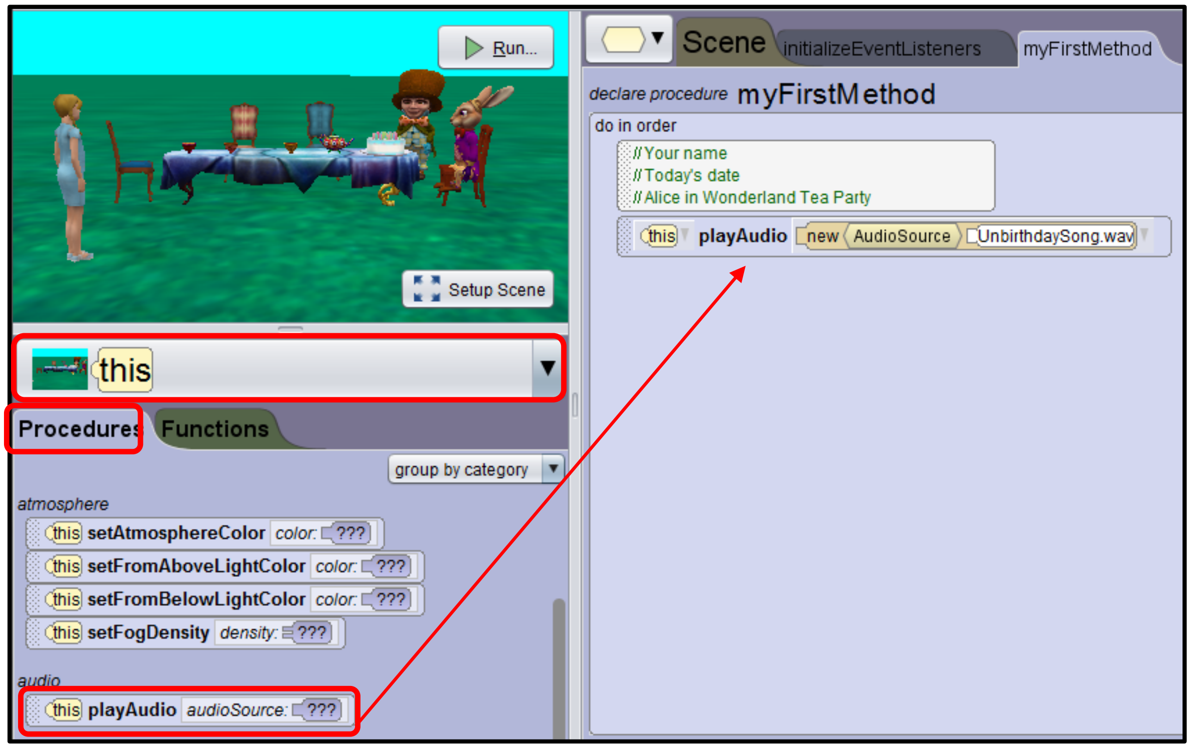
- Scene opens with the Mad Hatter and the March Hare gathered around a table with tea and a birthday cake
- The unbirthday song plays
- Alice approaches the table
- Alice tells the characters that she enjoyed their singing
- They tell her that nobody ever compliments their singing and insist that she has a cup of tea
- She apologizes for interrupting their birthday party
- They explain that it isn't their birthday; it is their "unbirthday"
- Alice then asks them to explain an "unbirthday"
- They then tell her that everyone has 364 "unbirthdays" each year
- Alice realizes that it is her "unbirthday" too

4. We need to save the new version of this file in the **Chapter1Exercises** folder instead of the Chapter0Exercises folder. Click on **File** from the menu, then **Save As**, locate the **Chapter1Exercises** folder, and save this file as **TeaParty**.

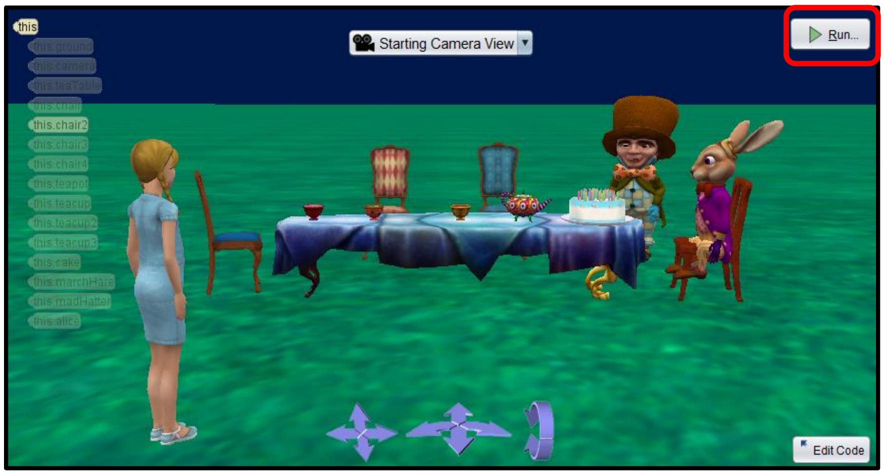
5. Drag the **//comment** block to the editor and enter your comments. You need to put your name, the date, and a description of the program in comments at the top of all of your programs.


```
declare procedure myFirstMethod on class Scene
do in order
    // Your Name
    // Today's Date
    // Alice in Wonderland Tea Party
```

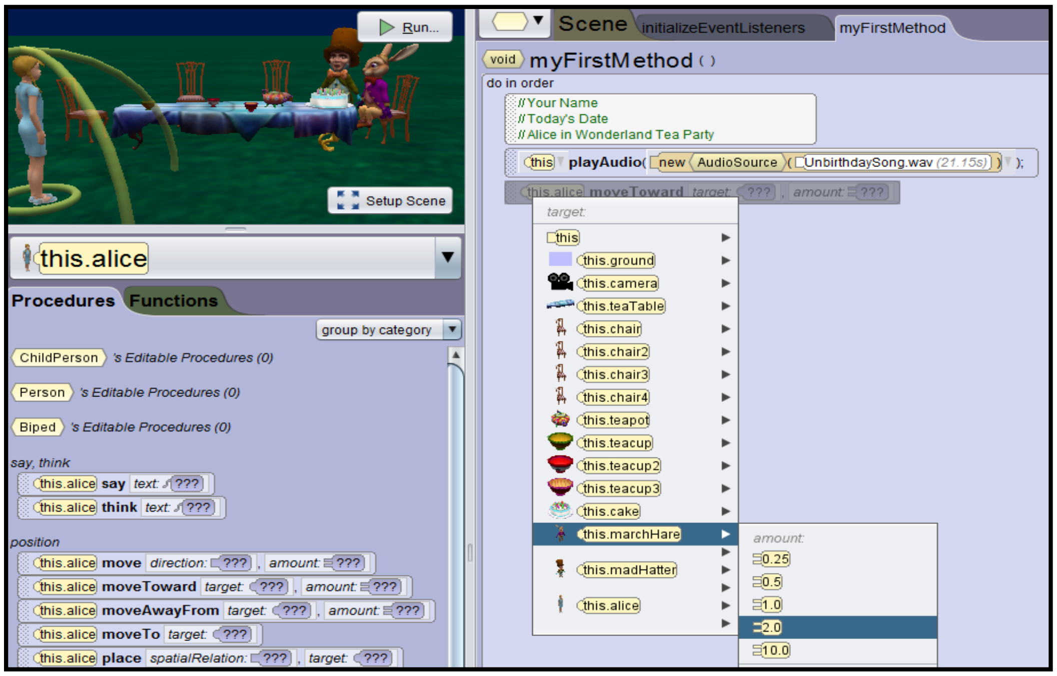
6. Click on the **this**, make sure that the **Procedures** tab is selected, drag the **playAudio** method to the editor, select **Import Audio**, use the **Unbirthday Song** file *(located in your Data_Files folder: http://iws.collin.edu/tdaly/book2/)*. Procedural methods are actions that objects can do. The word “this” refers to the scene and we are telling the scene to play the audio.


```
declare procedure myFirstMethod
do in order
    // Your name
    // Today's date
    // Alice in Wonderland Tea Party
    this playAudio new AudioSource UnbirthdaySong.wav
```
atmosphere
this setAtmosphereColor color: ???
this setFromAboveLightColor color: ???
this setFromBelowLightColor color: ???
this setFogDensity density: ???
audio
this playAudio audioSource: ???

7. We need to test the program by clicking the **Run** button to play the animation. You should hear the song play but nothing else happens yet.



8. Next, we will select alice, make sure that the procedures tab is selected, and drag the **moveToward method** onto the editor underneath the playAudio method. When you release the mouse, you will be prompted to select the target that you want alice to move toward and the amount that you want her to move. Select **marchHare** as the target and **2.0** as the amount. If you wanted a number that isn't on the list, you would select Custom DecimalNumber and type in your own number. These choices (target and amount) are known as arguments in programming. Run the animation to see if alice moves toward the marchHare; you will have to wait until the song finishes to see her move. The program happens in order. The next line doesn't execute until the previous line is finished.

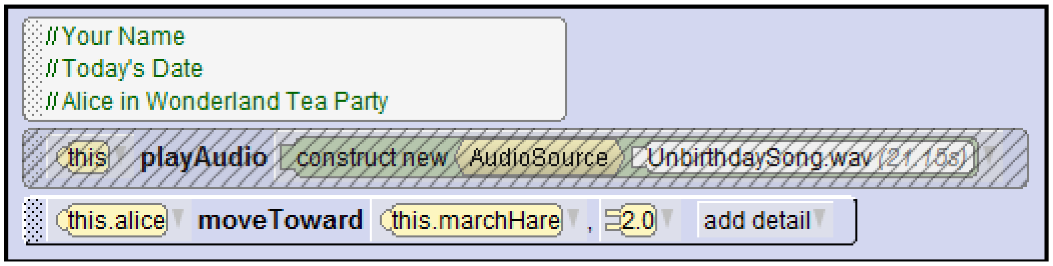


```
void myFirstMethod ( )
do in order
    // Your Name
    // Today's Date
    // Alice in Wonderland Tea Party
    this playAudio( new AudioSource( UnbirthdaySong.wav (21.15s) ) );
    this.alice moveToward target: ??? , amount: ???
```

```
this.alice say text: ???
this.alice think text: ???
this.alice move direction: ??? , amount: ???
this.alice moveToward target: ??? , amount: ???
this.alice moveAwayFrom target: ??? , amount: ???
this.alice moveTo target: ???
this.alice place spatialRelation: ??? , target: ???
```

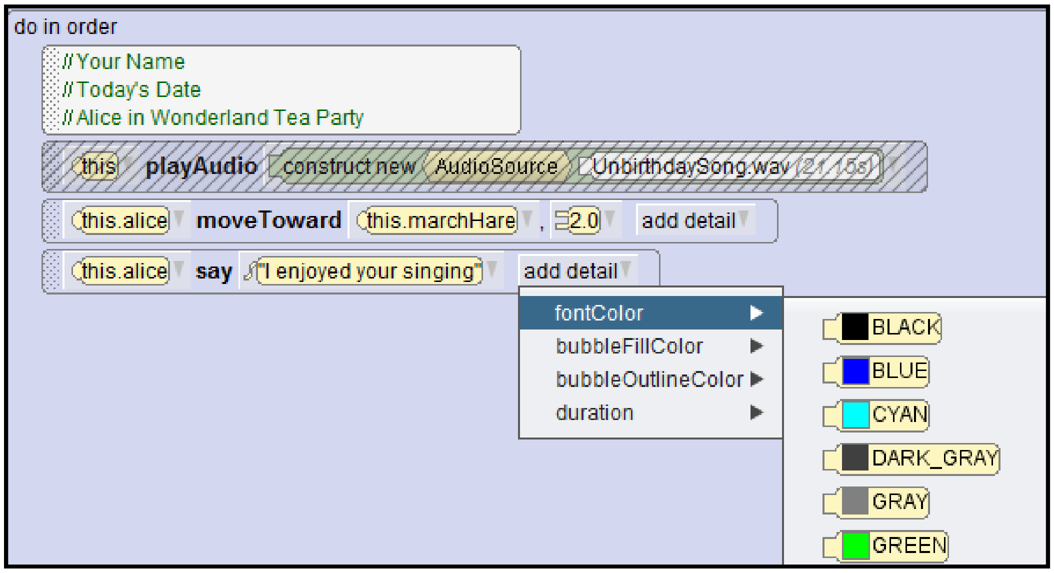
9. If you don't want to wait for the song to finish every time you want to test out your animation, you can disable lines of code and enable them later. To do this, you would

need to right click on the **playAudio** line and **uncheck Is Enabled**. You will see the line will now have gray lines over it.



```
// Your Name
// Today's Date
// Alice in Wonderland Tea Party
this playAudio construct new AudioSource UnbirthdaySong.wav (21.15s)
this.alice moveToward this.marchHare , 2.0 add detail
```

10. Now, we want alice to praise their singing. You will need to select alice and then drag the **say** method onto the editor underneath the moveToward method. When you do this, you will be prompted (argument) to enter the text of what you want alice to say. You should select **Custom TextString…** and then enter **I enjoyed your singing**. You can select **add detail** if you want to make adjustments such as text color, speech bubble color, outline color, or the duration that the bubble stays on the screen. You can leave the default settings if you want. The duration is defaulted to 1 second.



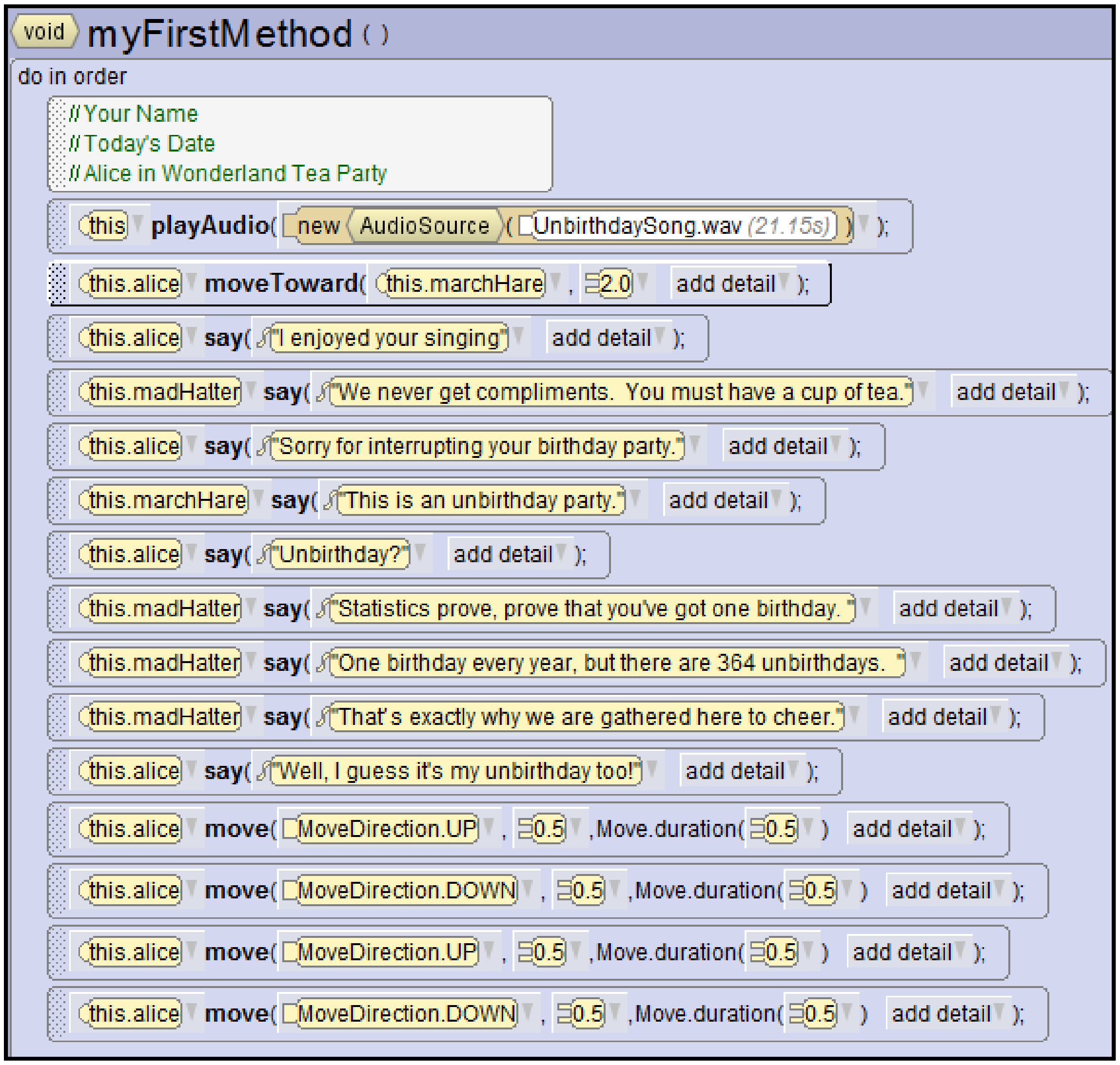
```
do in order
// Your Name
// Today's Date
// Alice in Wonderland Tea Party
this playAudio construct new AudioSource UnbirthdaySong.wav (21.15s)
this.alice moveToward this.marchHare , 2.0 add detail
this.alice say "I enjoyed your singing" add detail
```

11. Create the following dialog between the characters:

madHatter – We never get compliments, you must have a cup of tea.
alice – Sorry for interrupting your birthday party.
marchHare – This is an unbirthday party.
alice – Unbirthday?
madHatter - Statistics prove, prove that you've got one birthday. One birthday every year, but there are 364 unbirthdays. That's exactly why we are gathered here to cheer.
alice – Well I guess it's my unbirthday too!

12. Have Alice joyously jump up and down at the end. If you want her to jump at a faster pace, you can change the duration to be 0.5 seconds instead of 1 second (Click the *add detail* drop down to change the duration.

13. To test the full program with the song, you will need to **enable** the **playAudio** method. Right click on the **playAudio** line and **check Is Enabled**. The grey lines through the playAudio method should disappear.



```
void myFirstMethod ( )
do in order
    //Your Name
    //Today's Date
    //Alice in Wonderland Tea Party
    this playAudio( new AudioSource( UnbirthdaySong.wav (21.15s) ) );
    this.alice moveToward( this.marchHare , 2.0 add detail );
    this.alice say( "I enjoyed your singing" add detail );
    this.madHatter say( "We never get compliments.  You must have a cup of tea." add detail );
    this.alice say( "Sorry for interrupting your birthday party." add detail );
    this.marchHare say( "This is an unbirthday party." add detail );
    this.alice say( "Unbirthday?" add detail );
    this.madHatter say( "Statistics prove, prove that you've got one birthday. " add detail );
    this.madHatter say( "One birthday every year, but there are 364 unbirthdays. " add detail );
    this.madHatter say( "That's exactly why we are gathered here to cheer." add detail );
    this.alice say( "Well, I guess it's my unbirthday too!" add detail );
    this.alice move( MoveDirection.UP , 0.5 ,Move.duration( 0.5 ) add detail );
    this.alice move( MoveDirection.DOWN , 0.5 ,Move.duration( 0.5 ) add detail );
    this.alice move( MoveDirection.UP , 0.5 ,Move.duration( 0.5 ) add detail );
    this.alice move( MoveDirection.DOWN , 0.5 ,Move.duration( 0.5 ) add detail );
```

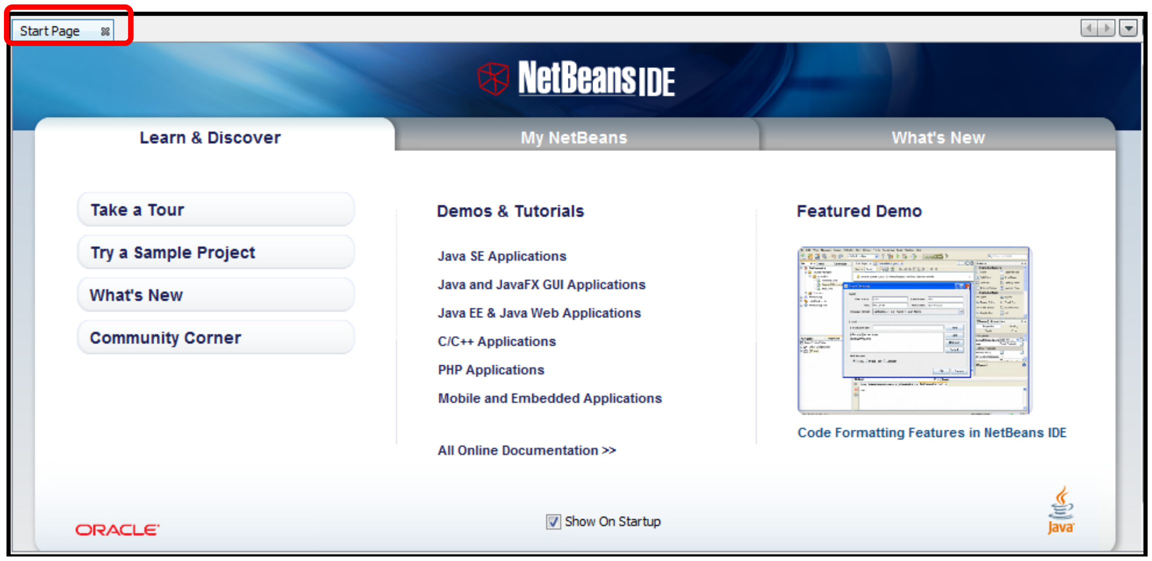
14. Challenge: Make this your own by adjusting it how you want. For example, you may want to challenge yourself and make the Mad Hatter and the March Hare dance to the unbirthday song. This would require that you use the **Do Together** block to make the dancing happen while the song is playing.

15. Save your work and exit Alice.

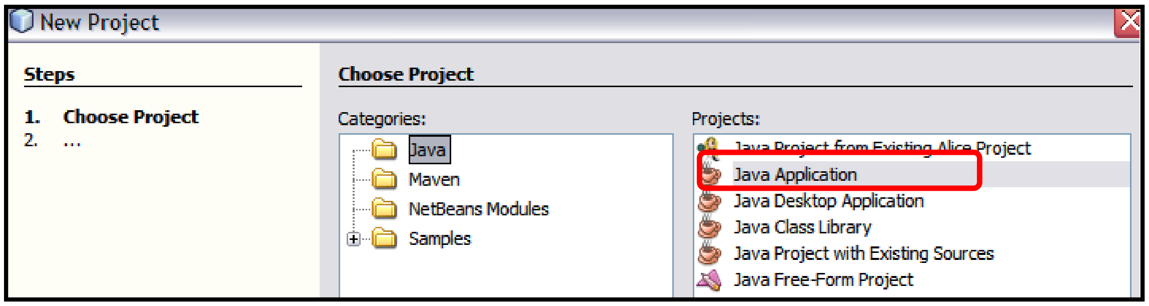
Exercise 2: Compiling and Executing a Java Program

1. Open up the NetBeans environment.

2. You can close the Start Page. The tutorials provided in the Start Page can be confusing for a first timer.

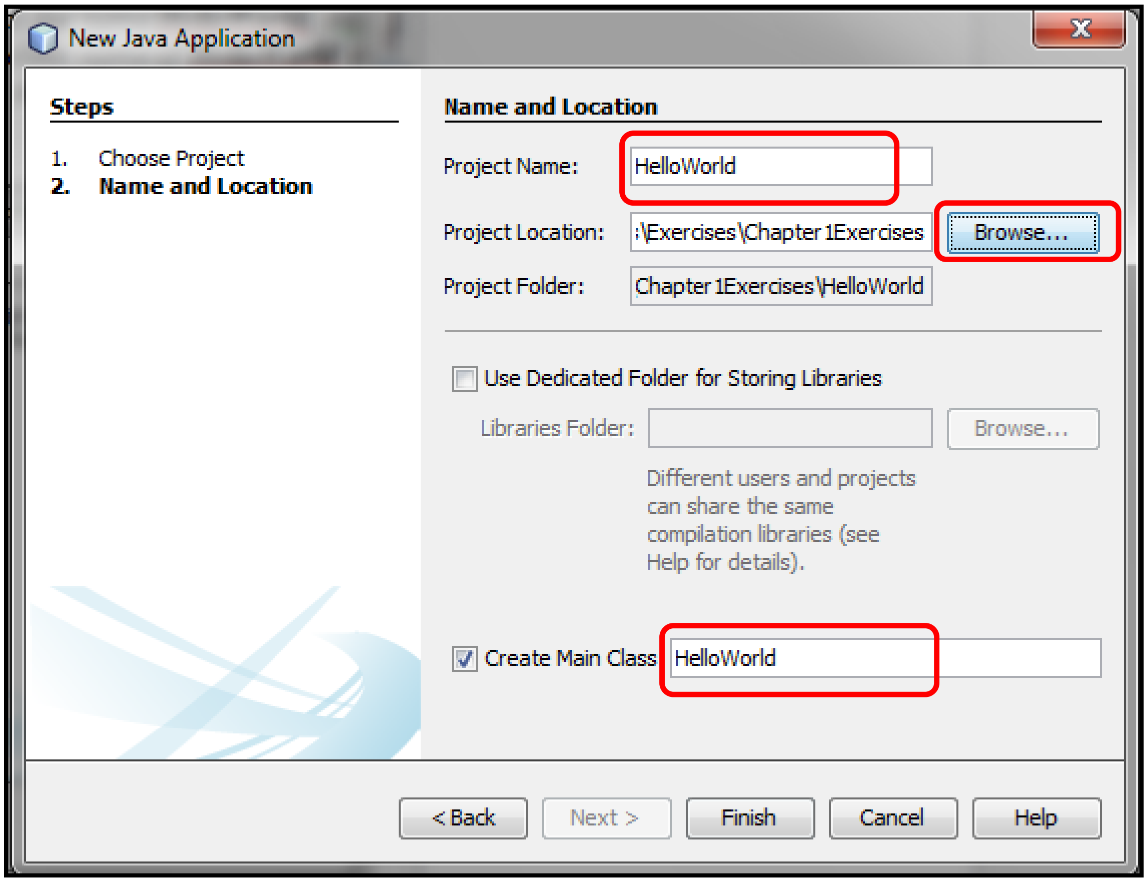


3. Select the **File** menu and then choose **New Project**. Then choose **Java Application** as shown below.



4. Click **Next**. Name your NetBeans project, select the **location** of where you would like to save your file, give your file *(Main Class)* a name *(make this name the same as your project name)*, and click **finish**. Although it is not necessary, we are going to name our projects and Java files *(main class)* have the same name. Therefore, make sure that the top and bottom boxes have the same name. NetBeans automatically will try to name your file (main class) helloworld.HelloWorld. **Erase the helloworld.** that NetBeans inserts before your file name. Make sure it looks like the following screenshot. Capitalization is important.

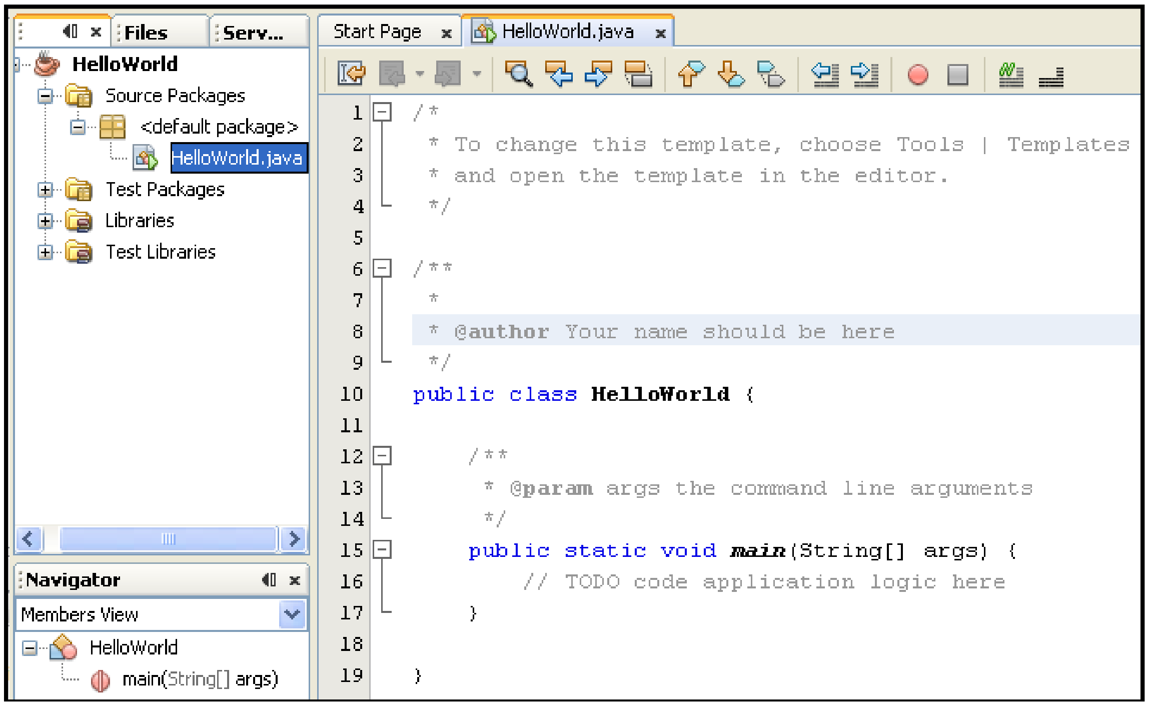
   *Project the name **HelloWorld**. (no spaces)*
   *Main class name of **HelloWorld** (no spaces)*
   *Select the **Location** of where you would like to save your NetBeans project*



Project Name: HelloWorld
Project Location: :\Exercises\Chapter 1Exercises
Project Folder: Chapter 1Exercises\HelloWorld
Use Dedicated Folder for Storing Libraries
Libraries Folder:
Different users and projects can share the same compilation libraries (see Help for details).
Create Main Class HelloWorld

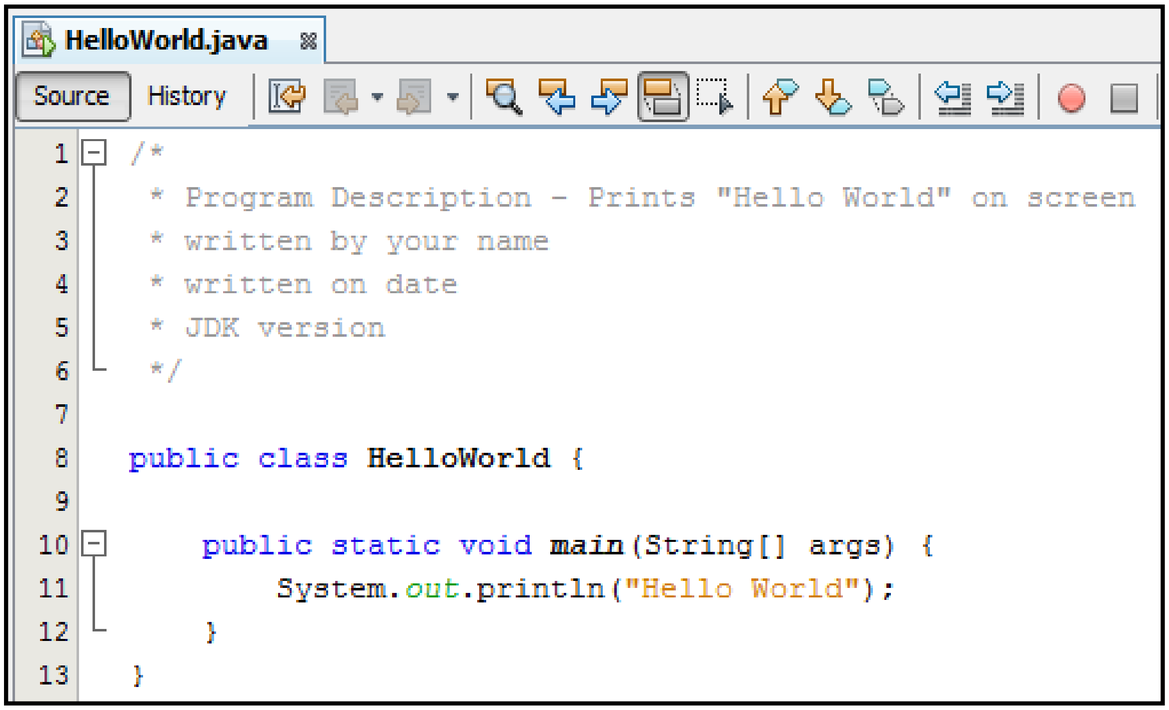
5. If line numbers are not showing, click **View** from the menu, then **Show Line Numbers.**

6. Your project should look as follows. HelloWorld.java is the file that we will be working with. *(Note: If your code has a package statement on line 5, then exit NetBeans and delete the project folder and do step 4 again. Make sure it looks like the screen shot provided. Look carefully at the textbox next to the Create Main Class label in the New Java Application dialog. For your information a package in Java is a folder. The package line indicates that you put your file in a folder when you created the project. We will not be using package statements in this text.)*



```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

/**
 *
 * @author Your name should be here
 */
public class HelloWorld {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
    }

}
```

7. Type in the following Java program. You will need to delete some comment lines and add some lines. Be careful to make your program look exactly as shown below. Try to keep your statements on the same lines as those shown below and also try to use the same approximate indentation to make your program more understandable. Change line 3 to be your name, line 4 to be today's date, and line 5 to be the JDK version that you are using *(You can check to see what version of the JDK that you have by clicking* ***Help*** *from the menu and then* ***About*** *in the NetBeans environment. It will have* ***Java:*** *and then a number, this is your JDK version. You do not need the number after the underscore).*



```
 1  /*
 2   * Program Description - Prints "Hello World" on screen
 3   * written by your name
 4   * written on date
 5   * JDK version
 6   */
 7
 8  public class HelloWorld {
 9
10      public static void main(String[] args) {
11          System.out.println("Hello World");
12      }
13  }
```

What does the above program do? You will not understand everything about this program YET. However, here is a brief explanation line by line:

1-6) Lines 1-6 are known as comments. Comments are used to document code so that other people reading our code can understand our logic. Comments are useful for adding extra information to our programs that we don't necessarily want to show up in the output of our program such as: author, date, JDK used, program description, etc. Also, it is a good idea to comment your programs extensively when you are just starting out so that you have well-documented examples. This is a multi-line comment which is represented by a /* at beginning of comment and */ to end the multi-line comment.

7) Blank line for readability purposes. Does nothing. (not necessary)

8) States this will be a public program called HelloWorld. Class names should begin with a capital letter. Be careful of capitalization in Java programs.

9) Blank line for readability purposes. Does nothing. (not necessary)

10) This is the main method declaration in this Java application. Every Java application must contain a main method and it must always be public static. The arguments for a method always appear in parentheses. In this case, the argument is a String array called args. The variable name of args can be whatever the programmer wants it to be, but most programmers use the variable name of args. The square brackets appearing after the word String are found to the right of the "P" key on keyboard.

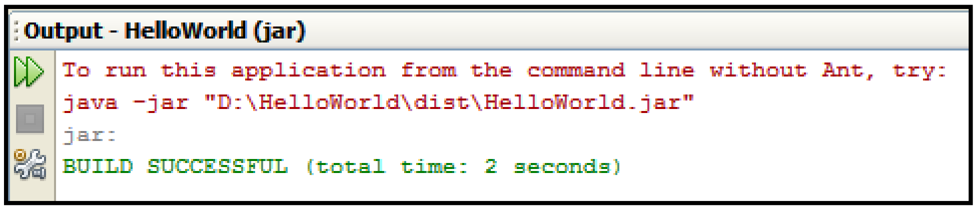
11) The statement of **System.out.println("Hello World");** prints Hello World to the screen and positions the insertion point on the next line. *System* is a Java class in the library and the *out* object is the screen. Java is case sensitive so be careful of capitalization. Java uses a punctuation method of class-dot-object-dot-method syntax. All methods have arguments in parenthesis which is a way of telling a method from a variable. This println method has an argument of a literal string of "Hello World". All Java statement lines will end with a semicolon.
Note: The next to last character in "println" is a lowercase L, not the number 1.

12) A right curly brace ends the main method. It is important to balance all your left and right curly braces and left and right parentheses in all Java programs. The right curly brace is found 2 keys to the right of the P key on keyboard.

13) A right curly brace to end the program.

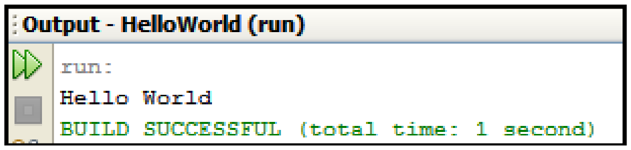
8. This Java program needs compiled. Compiling a program will have the computer look at each line of your program for syntax errors such as typos, mispunctuation, etc. To compile your program, click on **Run** from the File menu, then **Build Project**. The compiler will check this file for syntax errors and let you know on what lines you made errors. Errors (along with line numbers of errors) will list in bottom panel of the screen. If you have errors, correct your typos on the top of the screen and compile again. Make sure you adjust the bottom output panel large enough to see your errors and your output.

9. If there are no compilation errors (denoted by the words BUILD SUCCESSFUL), the compiler will convert this Java program into a bytecode file called ***HelloWorld.class*.** This bytecode file is a generic file that may be used on any operating system. This file is located under the **project** folder, under the **build** folder, and in the **classes** folder.



```
To run this application from the command line without Ant, try:
java -jar "D:\HelloWorld\dist\HelloWorld.jar"
jar:
BUILD SUCCESSFUL (total time: 2 seconds)
```

10. Once compiled and you have a bytecode file (.*class* file extension), you are ready to have the Java interpreter execute your Java program. To execute your first Java program, you will click on **Run** from the NetBeans menu, then **Run Project**. “Hello World” should be displayed in the output window as shown below:

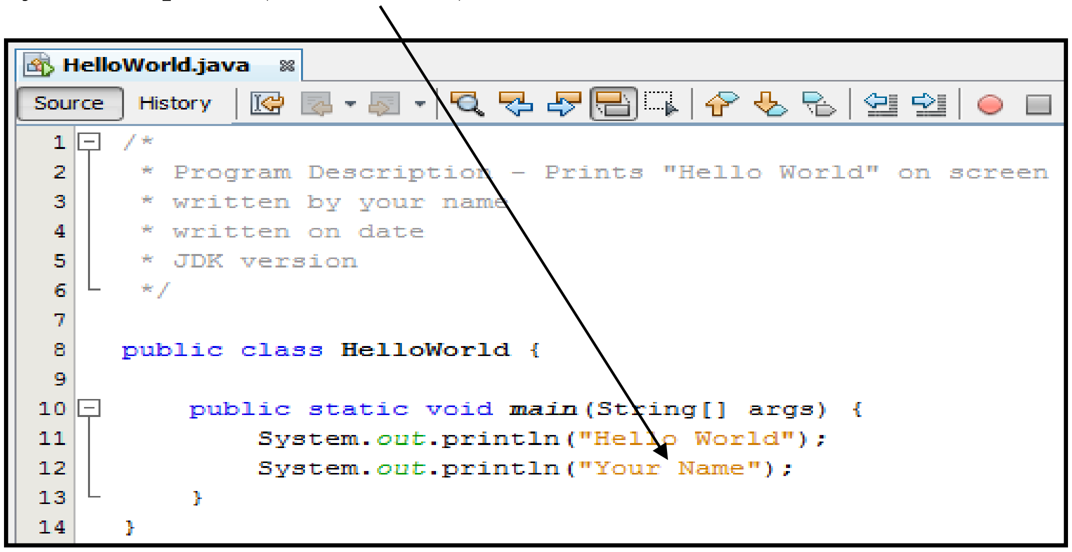


```
run:
Hello World
BUILD SUCCESSFUL (total time: 1 second)
```

11. The process you have seen so far is typing a Java program into NetBeans, compiling a Java program, and executing the Java program. This is the process that you will be going through over and over as you progress through Java. The output that you have at bottom of screen is simply the computer displaying the words "Hello World".

12. Now, let's adjust the Java program. Add the following line as shown in the diagram below. *(Note: if you type **sout** and hit the **tab key**, it will type the **System.out.println(" ");** line for you.)*

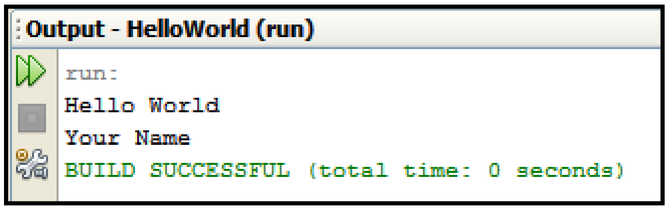
System.out.println("Your name");



```
/*
 * Program Description - Prints "Hello World" on screen
 * written by your name
 * written on date
 * JDK version
 */

public class HelloWorld {

    public static void main(String[] args) {
        System.out.println("Hello World");
        System.out.println("Your Name");
    }
}
```

13. Now, save the new version of the program by clicking on **Save** from the File menu. (**DO NOT** click "Save As" and save this outside the project folder. NetBeans has a file structure and you cannot pull your files out of this structure or else NetBeans will not open them in the future. Compile the program (**Run** menu and then **Build Project**).

14. Execute the program (**Run** menu and then **Run Project**). Your display window should look similar to the following:

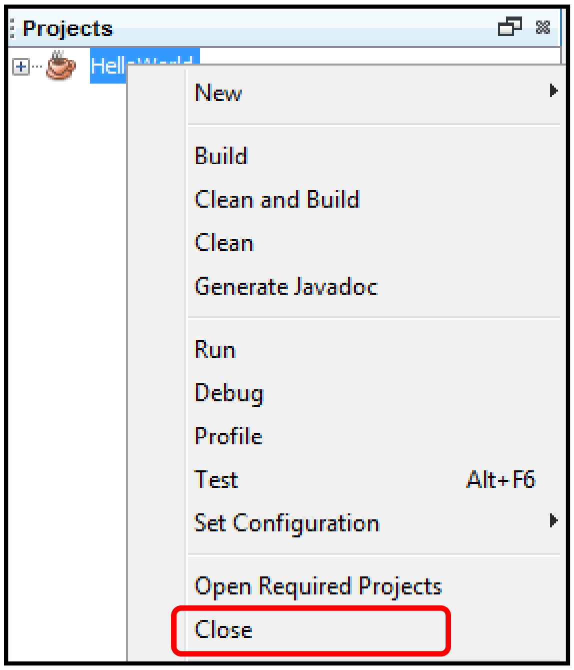


```
run:
Hello World
Your Name
BUILD SUCCESSFUL (total time: 0 seconds)
```

Note: If you are getting compiler errors at bottom of screen, please double check the capitalization, spelling, and punctuation.

15. To ensure that your code indentation is correct, you should always choose **Source** from the menu, then **Format**. Make sure that you compile your program (Run menu, Build Project) and run it (Run menu, Run Project).

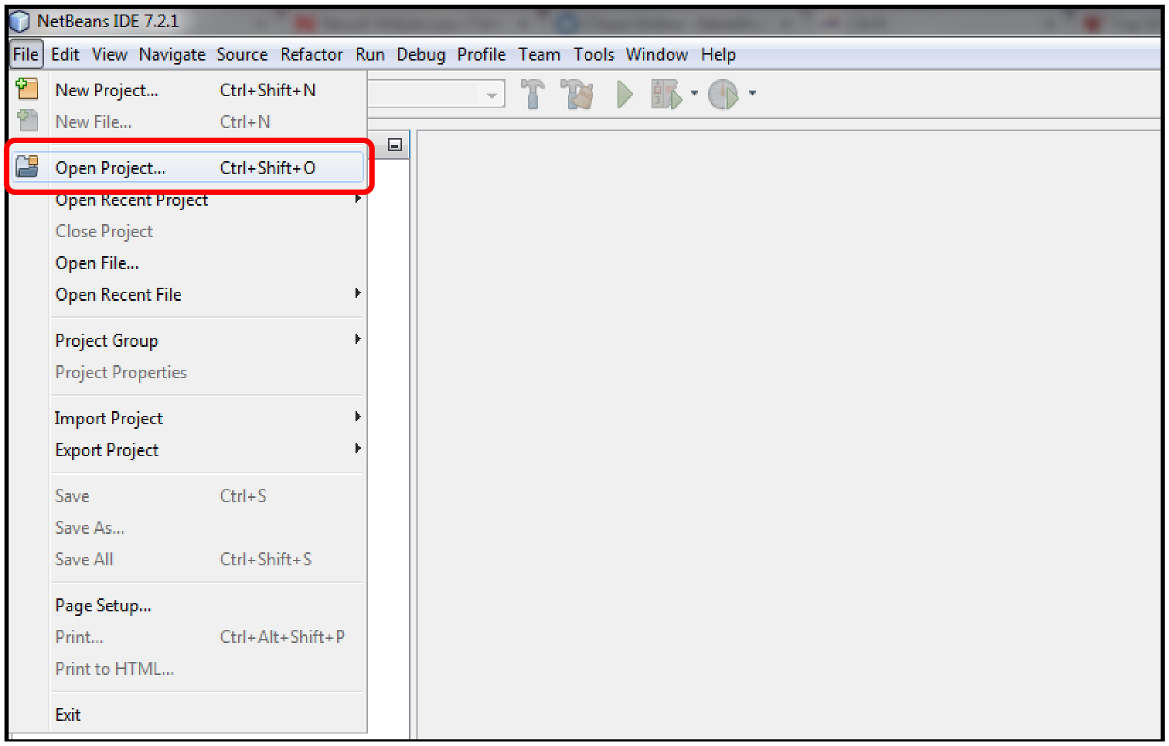
16. Close the project by right clicking on the project on the left pane and choosing **Close**.

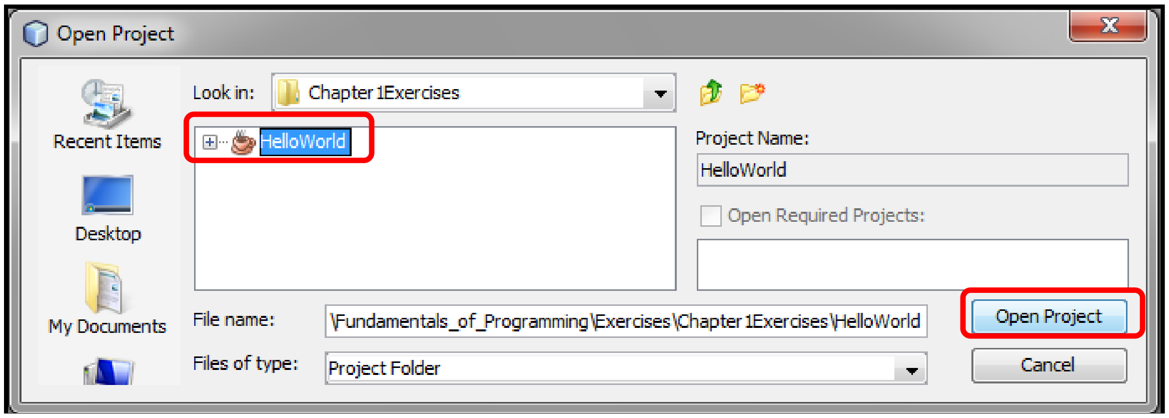


Exercise 3: Simple Java Debugging

1. Open up the NetBeans environment if you have exited.

2. Choose **File**, **Open Project…**, select the **HelloWorld** NetBeans project (should have a coffee cup next to your project), and click **Open Project**.





Look in: Chapter 1Exercises

HelloWorld

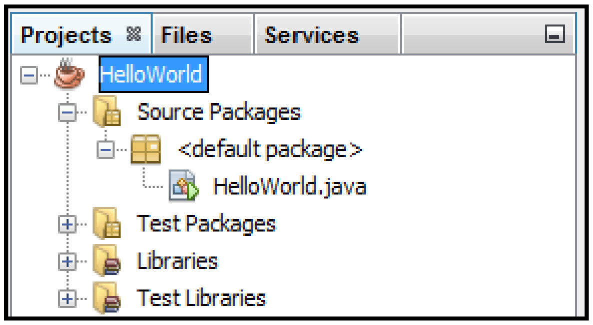
Project Name: HelloWorld

Open Required Projects:

File name: \Fundamentals_of_Programming\Exercises\Chapter 1Exercises\HelloWorld

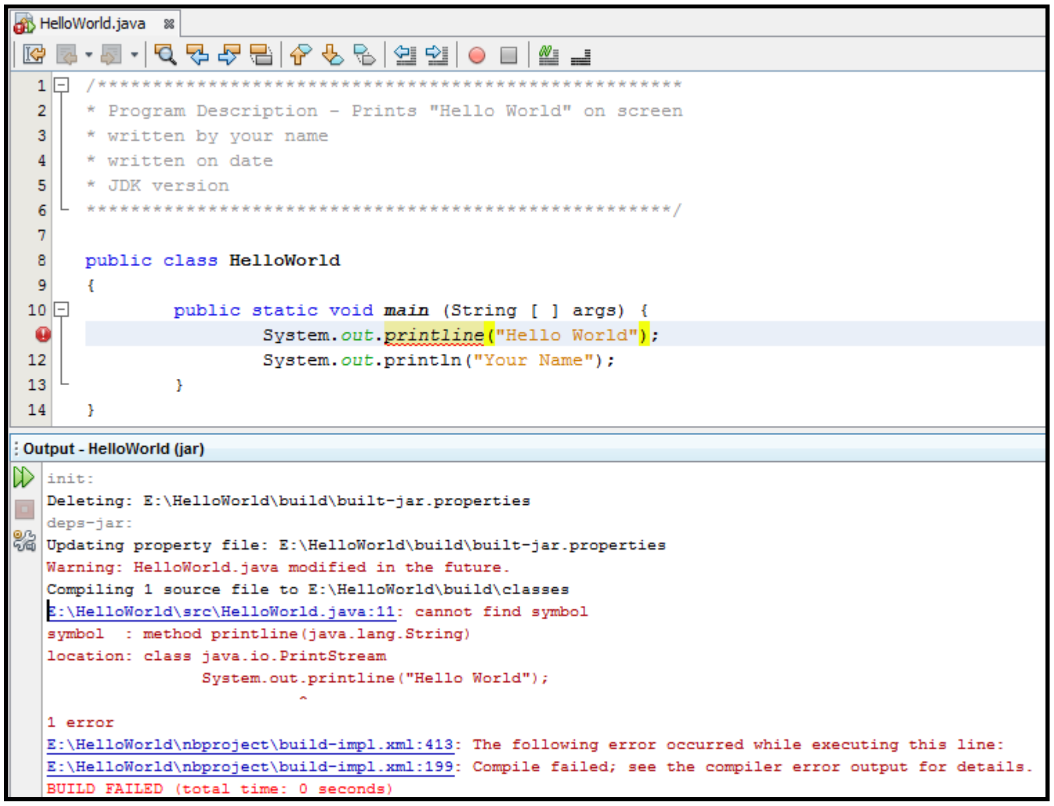
Files of type: Project Folder

3. Open your code, by expanding the **HelloWorld project folder**, then expanding the **Source Package folder**, then expanding the **default package folder** *(this is default since we did not name this folder when we created the project)* as shown below. You will need to double click on the **HelloWorld.java** file to open the code.



4. You will now purposely make some errors in your program. Change the spelling of **println** to be **printline** as follows:
   System.out.printline("Hello World");

5. Compile the program by clicking on **Run** menu and choosing **Build Project**. You should get an error at the bottom of your screen as follows:



```
/*********************************************************
 * Program Description - Prints "Hello World" on screen
 * written by your name
 * written on date
 * JDK version
 *********************************************************/

public class HelloWorld
{
        public static void main (String [ ] args) {
                System.out.printline("Hello World");
                System.out.println("Your Name");
        }
}
```

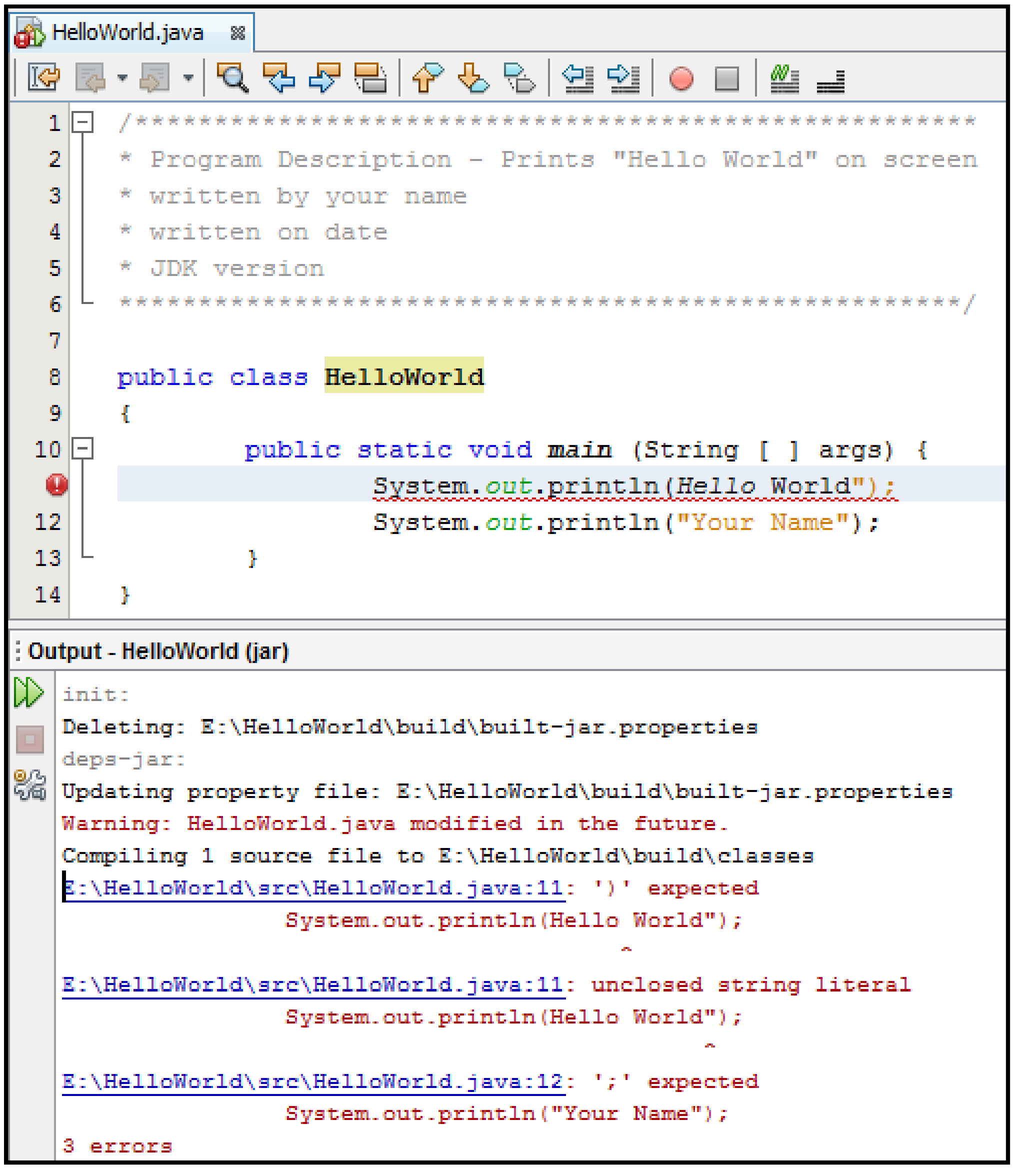
```
init:
Deleting: E:\HelloWorld\build\built-jar.properties
deps-jar:
Updating property file: E:\HelloWorld\build\built-jar.properties
Warning: HelloWorld.java modified in the future.
Compiling 1 source file to E:\HelloWorld\build\classes
E:\HelloWorld\src\HelloWorld.java:11: cannot find symbol
symbol  : method printline(java.lang.String)
location: class java.io.PrintStream
                System.out.printline("Hello World");
                          ^
1 error
E:\HelloWorld\nbproject\build-impl.xml:413: The following error occurred while executing this line:
E:\HelloWorld\nbproject\build-impl.xml:199: Compile failed; see the compiler error output for details.
BUILD FAILED (total time: 0 seconds)
```

6. It is telling you that there is an error on line 11. It could not find a method spelled as **printline**. Some errors will be obvious and those are the nice ones to solve. Change the word **printline** to be **println** so that this error is corrected.

7. Compile the program. You should be rid of all errors and it should say BUILD SUCCESSFUL. Lesson learned: Names must be spelled exactly as Java expects.

8. Change line 11 by deleting the opening set of double quotes around Hello World as follows: **System.out.println(Hello World");**

9. Compile the program. You should get 3 errors at the bottom of your screen:



```
 1  /*****************************************************
 2   * Program Description - Prints "Hello World" on screen
 3   * written by your name
 4   * written on date
 5   * JDK version
 6   *****************************************************/
 7
 8  public class HelloWorld
 9  {
10          public static void main (String [ ] args) {
                    System.out.println(Hello World");
12                  System.out.println("Your Name");
13          }
14  }
```

```
init:
Deleting: E:\HelloWorld\build\built-jar.properties
deps-jar:
Updating property file: E:\HelloWorld\build\built-jar.properties
Warning: HelloWorld.java modified in the future.
Compiling 1 source file to E:\HelloWorld\build\classes
E:\HelloWorld\src\HelloWorld.java:11: ')' expected
                System.out.println(Hello World");
                                        ^
E:\HelloWorld\src\HelloWorld.java:11: unclosed string literal
                System.out.println(Hello World");
                                              ^
E:\HelloWorld\src\HelloWorld.java:12: ';' expected
                System.out.println("Your Name");
3 errors
```

10. So, why would you get multiple error messages when you just made one mistake? This can happen. You may even get 15 errors for just one mistake. It really depends upon what mistake you make. In this case, it is saying it doesn't understand the Hello World and is saying it thinks it needs a parenthesis. This is not really the case. What it needs is a string enclosed in double quotes, but the error it shows is not real helpful. Now, insert the double quote back in front of the word *Hello*.

11. Compile the program. You should be rid of all errors. Lesson learned: The Java compiler doesn't always pinpoint the exact error. You must learn to look for errors anywhere on that line or previous line.

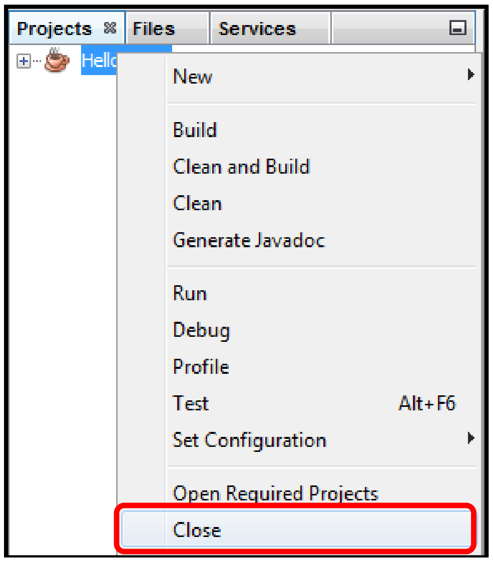
12. Not all errors are compilation errors. We now have a bug-free (no errors) Java program.

13. A programmer sometimes makes logic errors. Change line 11 to say:
    System.out.println("Helo Warld");
    Note: Hello and World are incorrectly spelled.

14. Compile the program. The program should get a compile with BUILD SUCCESSFUL. However, when this program is executed (RUN menu, then RUN MAIN PROJECT), you will not get the words displayed that you wanted. The reason that you get no errors in the compilation is because the words inside of double quotes can be anything. The computer has no idea what you are trying to accomplish. It will display anything that is in double quotes and doesn't check that part for incorrect spellings.

15. Please fix all of your errors and recompile. NetBeans automatically saves files when your project is compiled.

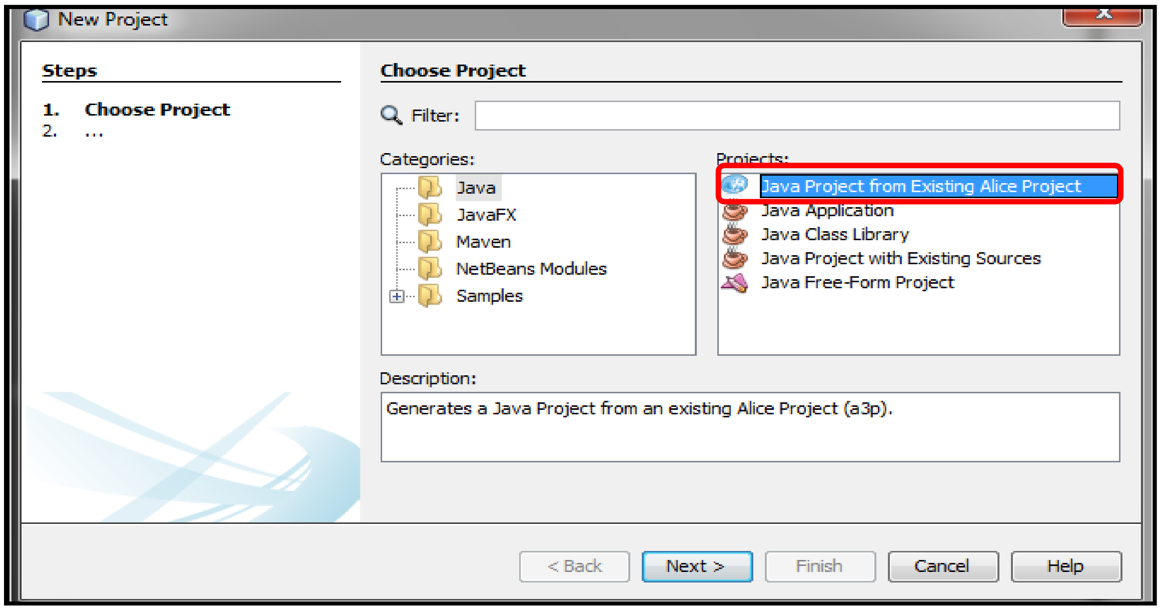
16. Close the HelloWorld project by right clicking on the project in the left pane as shown below:



Exercise 4: Transitioning Alice to Java (ongoing exercise)

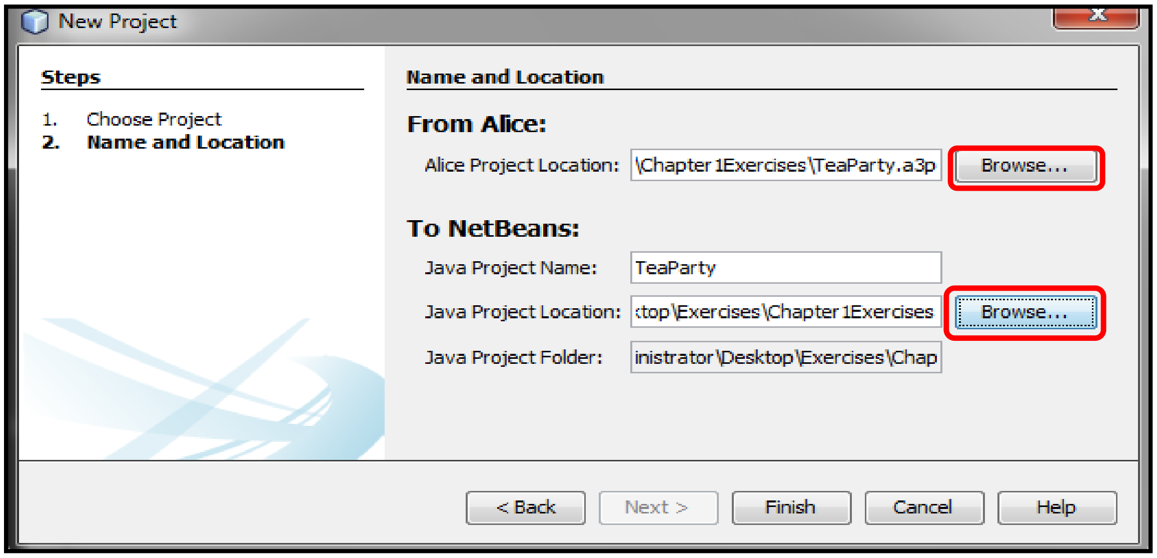
1. We already wrote the Alice code for the tea party in a previous exercise. Now let's see the Java code for this exercise using NetBeans. Open NetBeans if you have closed it.

2. Select **File** from the menu and then choose **New Project**. Then choose **Java Project from Existing Alice Project** as shown below. *(If this is not an option under the Java folder, then you need to install the Alice plugin for NetBeans from the install directions).*



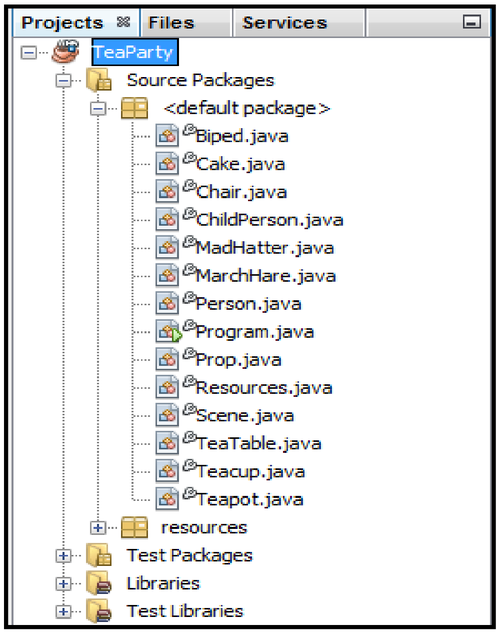
Description:
Generates a Java Project from an existing Alice Project (a3p).

3. Select **Next**, Click the **Browse** button next to the Alice world and find where you saved your Alice TeaParty file; it should be in the Chapter1Exercises folder. Click **Browse** to change the location of where it is going to save the new project. You should save this in the same place as your Alice project under the **Chapter1Exercises** folder. Click **Finish**. It may take a few minutes to pull up the Java files.

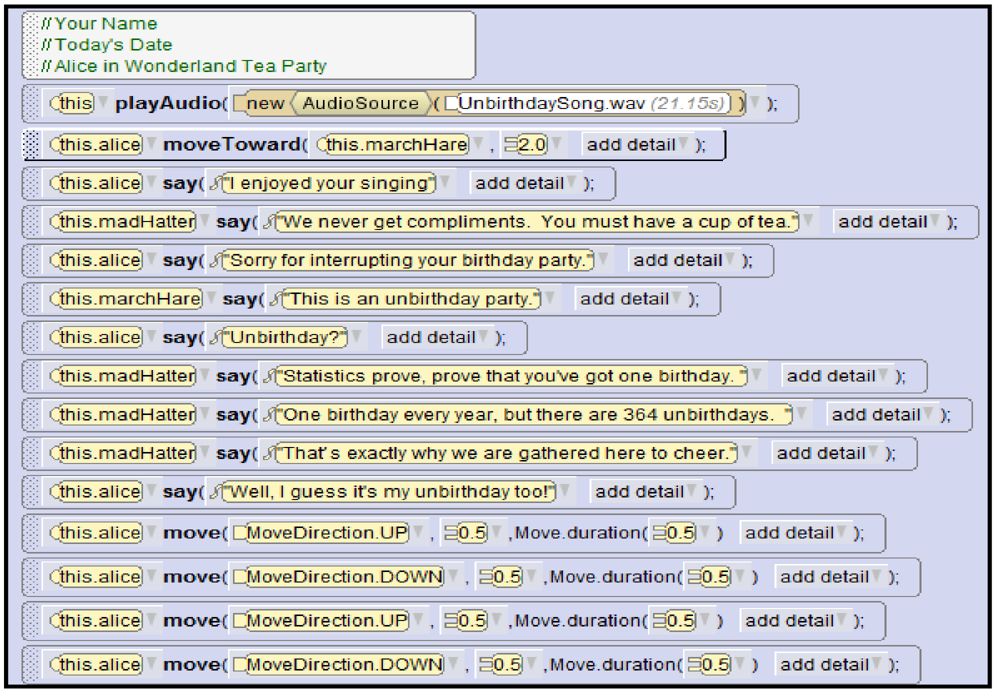


Alice Project Location: \Chapter1Exercises\TeaParty.a3p
Java Project Name: TeaParty
Java Project Location: :top\Exercises\Chapter1Exercises
Java Project Folder: inistrator\Desktop\Exercises\Chap

4. You should see the TeaParty project on the left hand side of the window. If you click the plus sign next to the TeaParty project, it will **expand** the **TeaParty folder** and you will see the folders inside of the project that are used to make the NetBeans project work. If you **expand** the **Source Packages folder** and the **default package folder**, you will see all of the Java files for your project. This may be overwhelming at first, but eventually it will make sense.

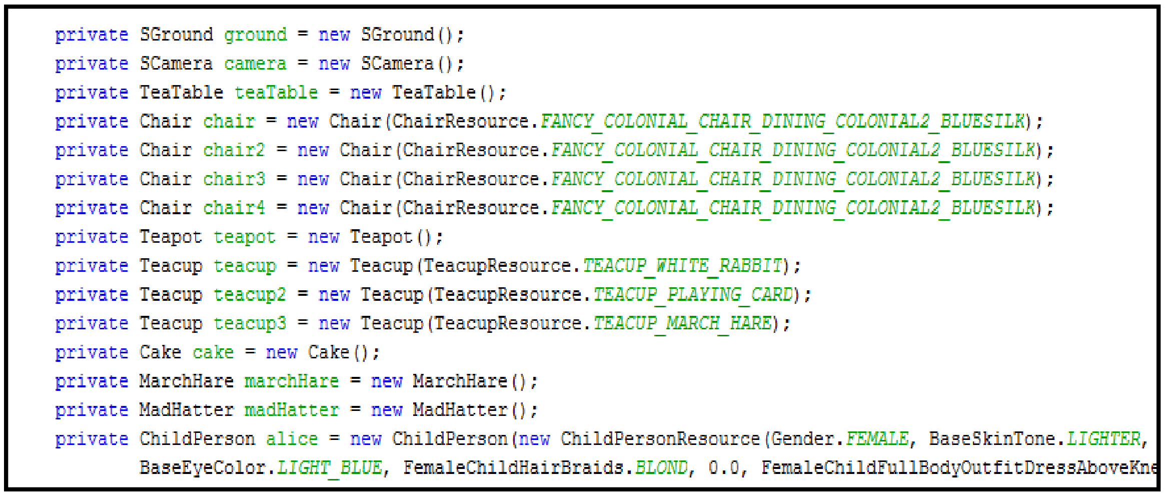


5. You should double click on the **Scene.java** file to open it up. This is where the code that we wrote in Alice will be. We wrote our Alice code in the myFirstMethod of the Scene tab. We are going to compare this Alice code with the NetBeans Java code.



```
//Your Name
//Today's Date
//Alice in Wonderland Tea Party
this playAudio( new AudioSource ( UnbirthdaySong.wav (21.15s) ) );
this.alice moveToward( this.marchHare , 2.0 add detail );
this.alice say( "I enjoyed your singing" add detail );
this.madHatter say( "We never get compliments.  You must have a cup of tea." add detail );
this.alice say( "Sorry for interrupting your birthday party." add detail );
this.marchHare say( "This is an unbirthday party." add detail );
this.alice say( "Unbirthday?" add detail );
this.madHatter say( "Statistics prove, prove that you've got one birthday. " add detail );
this.madHatter say( "One birthday every year, but there are 364 unbirthdays. " add detail );
this.madHatter say( "That's exactly why we are gathered here to cheer." add detail );
this.alice say( "Well, I guess it's my unbirthday too!" add detail );
this.alice move( MoveDirection.UP , 0.5 ,Move.duration( 0.5 ) add detail );
this.alice move( MoveDirection.DOWN , 0.5 ,Move.duration( 0.5 ) add detail );
this.alice move( MoveDirection.UP , 0.5 ,Move.duration( 0.5 ) add detail );
this.alice move( MoveDirection.DOWN , 0.5 ,Move.duration( 0.5 ) add detail );
```

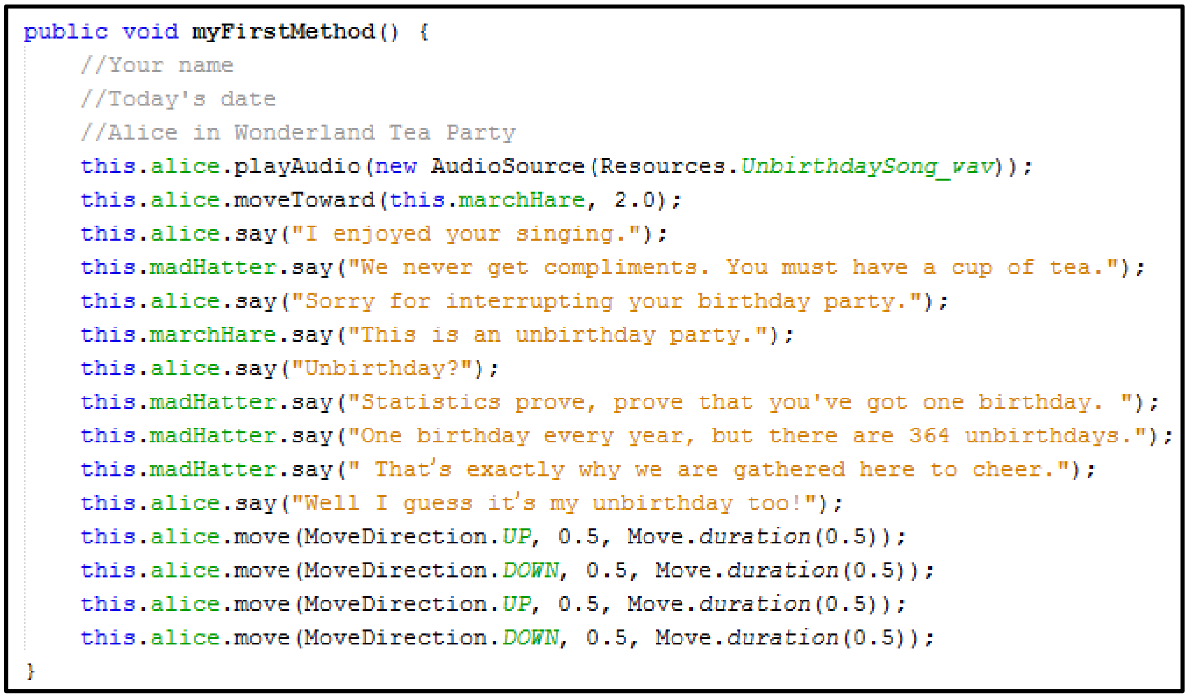
6. We are going to take a look at a few sections of this **Scene** file. The following code creates the objects for your scene.



```
private SGround ground = new SGround();
private SCamera camera = new SCamera();
private TeaTable teaTable = new TeaTable();
private Chair chair = new Chair(ChairResource.FANCY_COLONIAL_CHAIR_DINING_COLONIAL2_BLUESILK);
private Chair chair2 = new Chair(ChairResource.FANCY_COLONIAL_CHAIR_DINING_COLONIAL2_BLUESILK);
private Chair chair3 = new Chair(ChairResource.FANCY_COLONIAL_CHAIR_DINING_COLONIAL2_BLUESILK);
private Chair chair4 = new Chair(ChairResource.FANCY_COLONIAL_CHAIR_DINING_COLONIAL2_BLUESILK);
private Teapot teapot = new Teapot();
private Teacup teacup = new Teacup(TeacupResource.TEACUP_WHITE_RABBIT);
private Teacup teacup2 = new Teacup(TeacupResource.TEACUP_PLAYING_CARD);
private Teacup teacup3 = new Teacup(TeacupResource.TEACUP_MARCH_HARE);
private Cake cake = new Cake();
private MarchHare marchHare = new MarchHare();
private MadHatter madHatter = new MadHatter();
private ChildPerson alice = new ChildPerson(new ChildPersonResource(Gender.FEMALE, BaseSkinTone.LIGHTER,
        BaseEyeColor.LIGHT_BLUE, FemaleChildHairBraids.BLOND, 0.0, FemaleChildFullBodyOutfitDressAboveKne
```

7. The **myFirstMethod** is where the code that we added to Alice will be. The word **this** refers to the scene. The first statement is saying to use the playAudio method on the scene. The second statement is telling Java to use the moveToward method on alice from the scene. The marchHare from this scene is the first argument (what you want alice to move towards) and the 2.0 is the second argument (the distance that you want her to move towards the marchHare).

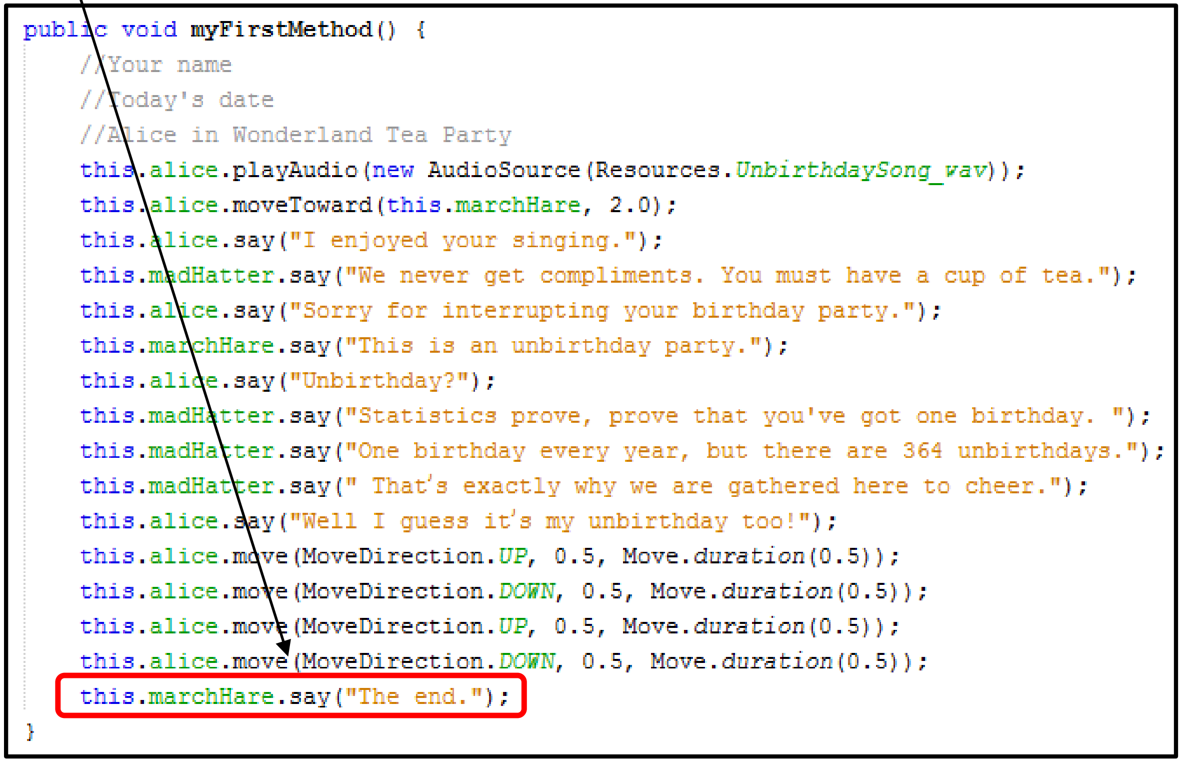
Note: If you are using a JDK earlier than 1.8, the Alice comments will not transfer into NetBeans.



```
public void myFirstMethod() {
    //Your name
    //Today's date
    //Alice in Wonderland Tea Party
    this.alice.playAudio(new AudioSource(Resources.UnbirthdaySong_wav));
    this.alice.moveToward(this.marchHare, 2.0);
    this.alice.say("I enjoyed your singing.");
    this.madHatter.say("We never get compliments. You must have a cup of tea.");
    this.alice.say("Sorry for interrupting your birthday party.");
    this.marchHare.say("This is an unbirthday party.");
    this.alice.say("Unbirthday?");
    this.madHatter.say("Statistics prove, prove that you've got one birthday. ");
    this.madHatter.say("One birthday every year, but there are 364 unbirthdays.");
    this.madHatter.say(" That's exactly why we are gathered here to cheer.");
    this.alice.say("Well I guess it's my unbirthday too!");
    this.alice.move(MoveDirection.UP, 0.5, Move.duration(0.5));
    this.alice.move(MoveDirection.DOWN, 0.5, Move.duration(0.5));
    this.alice.move(MoveDirection.UP, 0.5, Move.duration(0.5));
    this.alice.move(MoveDirection.DOWN, 0.5, Move.duration(0.5));
}
```

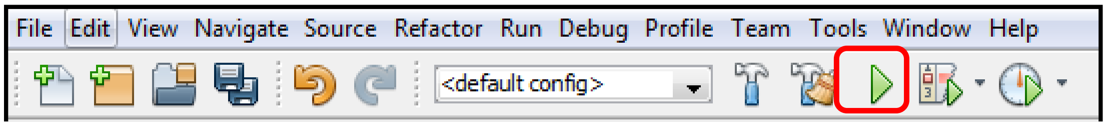
8. Add the following line of code to the end of myFirstMethod. You can see below that the end of the myFirstMethod is after the last move statement and before the ending curly brace for the method.

this.marchHare.say("The End!");

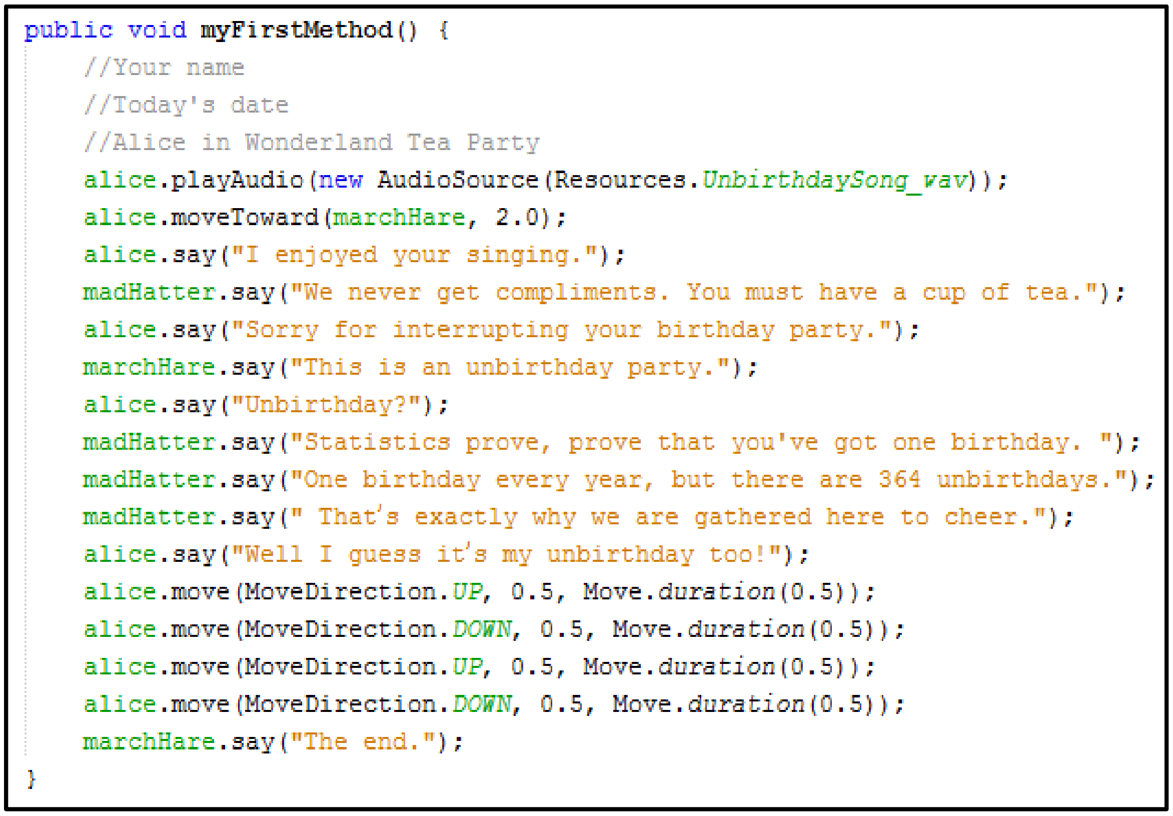


```
public void myFirstMethod() {
    //Your name
    //Today's date
    //Alice in Wonderland Tea Party
    this.alice.playAudio(new AudioSource(Resources.UnbirthdaySong_wav));
    this.alice.moveToward(this.marchHare, 2.0);
    this.alice.say("I enjoyed your singing.");
    this.madHatter.say("We never get compliments. You must have a cup of tea.");
    this.alice.say("Sorry for interrupting your birthday party.");
    this.marchHare.say("This is an unbirthday party.");
    this.alice.say("Unbirthday?");
    this.madHatter.say("Statistics prove, prove that you've got one birthday. ");
    this.madHatter.say("One birthday every year, but there are 364 unbirthdays.");
    this.madHatter.say(" That's exactly why we are gathered here to cheer.");
    this.alice.say("Well I guess it's my unbirthday too!");
    this.alice.move(MoveDirection.UP, 0.5, Move.duration(0.5));
    this.alice.move(MoveDirection.DOWN, 0.5, Move.duration(0.5));
    this.alice.move(MoveDirection.UP, 0.5, Move.duration(0.5));
    this.alice.move(MoveDirection.DOWN, 0.5, Move.duration(0.5));
    this.marchHare.say("The end.");
}
```

9. To test this animation, we will click on the **run** button on the menu in NetBeans. The run button is a shortcut way to save, compile, and run.


File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
<default config>

10. In this example the word **this** is optional (it refers to the scene). You can remove it if you find it confusing. Please see the code below. It is up to you if you want to leave it alone or remove it.



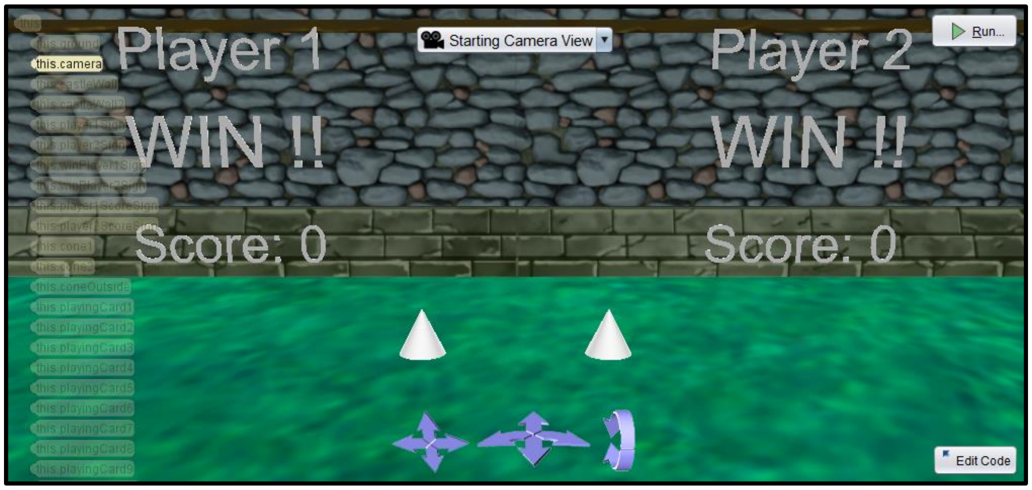
```
public void myFirstMethod() {
    //Your name
    //Today's date
    //Alice in Wonderland Tea Party
    alice.playAudio(new AudioSource(Resources.UnbirthdaySong_wav));
    alice.moveToward(marchHare, 2.0);
    alice.say("I enjoyed your singing.");
    madHatter.say("We never get compliments. You must have a cup of tea.");
    alice.say("Sorry for interrupting your birthday party.");
    marchHare.say("This is an unbirthday party.");
    alice.say("Unbirthday?");
    madHatter.say("Statistics prove, prove that you've got one birthday. ");
    madHatter.say("One birthday every year, but there are 364 unbirthdays.");
    madHatter.say(" That’s exactly why we are gathered here to cheer.");
    alice.say("Well I guess it’s my unbirthday too!");
    alice.move(MoveDirection.UP, 0.5, Move.duration(0.5));
    alice.move(MoveDirection.DOWN, 0.5, Move.duration(0.5));
    alice.move(MoveDirection.UP, 0.5, Move.duration(0.5));
    alice.move(MoveDirection.DOWN, 0.5, Move.duration(0.5));
    marchHare.say("The end.");
}
```

All of this code may seem overwhelming at first, but by the end of this class you will understand most of this code.

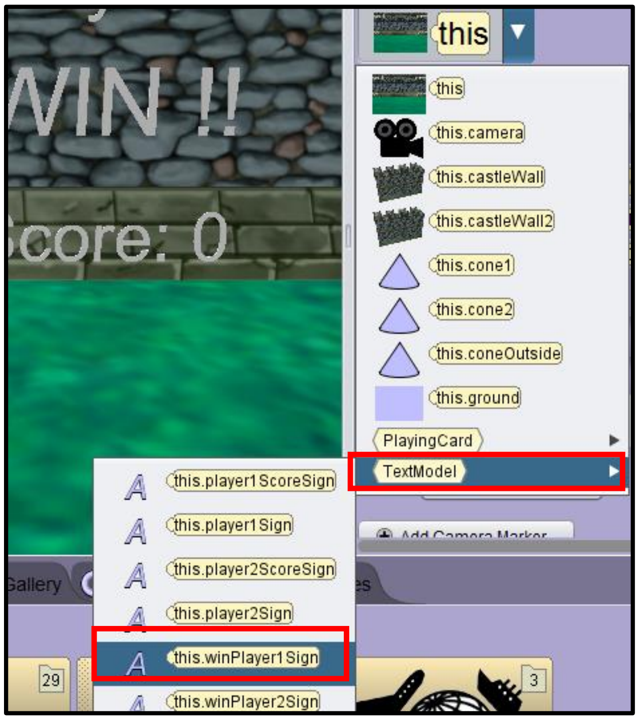
11. Run your program before exiting to save and test your changes.

Exercise 5: Alice Card Game Coding (ongoing exercise)

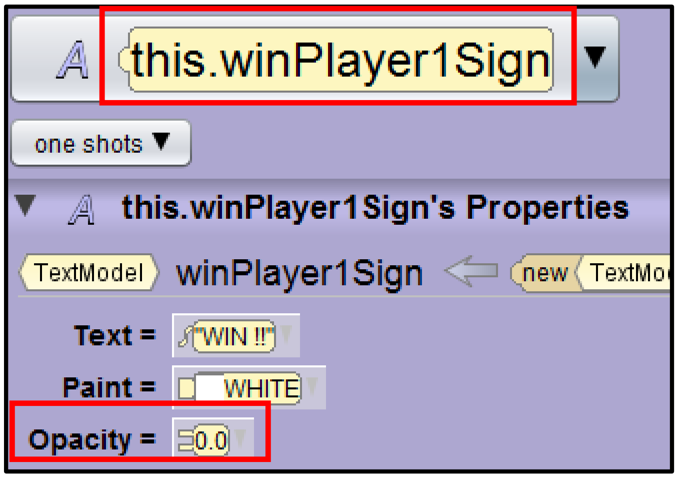
1. Open up Alice 3. Open the file named **CardGame** from the Chapter0Exercises or **MyCardGame** from the Chapter 0 Assignments. In Chapter 0, you set up a Card Game scene as follows:



2. Go to Setup Scene. Some items will need to be adjusted before beginning the playing of the game. The two win signs and the cones need to be made invisible. Choose the **winPlayer1Sign** object on the right side of panel by choosing TextModel and then winPlayer1Sign as follows:



3. The WIN sign on left side of screen should be showing as selected. Change the **opacity** to be **0.0**



Text = "WIN !!"
Paint = WHITE
Opacity = 0.0

This will cause the WIN sign to disappear on left side.

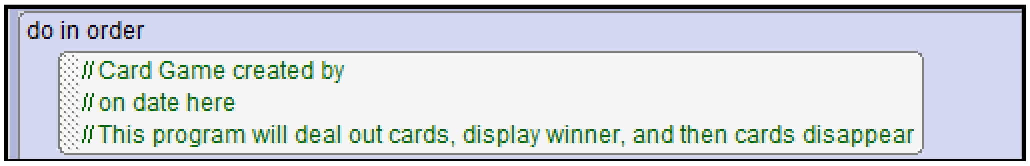
4. Select the **winPlayer2Sign** and set its **opacity** to **0.0**. Both WIN signs should now be invisible. Select the **cone1** object and set its **opacity** to **0.0** so that the left cone disappears. It is still there but invisible. Select the **cone2** object and set its opacity to **0.0** so that the right cone disappears. Our game is now ready for code. Click on EDIT CODE button to take you back to main screen.

5. Save the new version of this file in the **Chapter1Exercises** folder instead of the Chapter0Exercises folder. Click on **File** from the menu, then **Save As**, locate the **Chapter1Exercises** folder, and save this file as **CardGame**.

6. Before writing the code for card game, we should first create a storyboard of what we wish to accomplish.

- Scene opens with the CardGame
- Player cards are dealt out onto screen
- Display winner
- Have these cards disappear
- New Player cards are dealt out on screen
- Deteremine and display winner
- Have these cards disappear

7. Every program you write should begin with comments for documentation. Drag the **//comment** block to the editor and enter your comments. You need to put your name, the date, and a description of the program at the top of all of your programs.

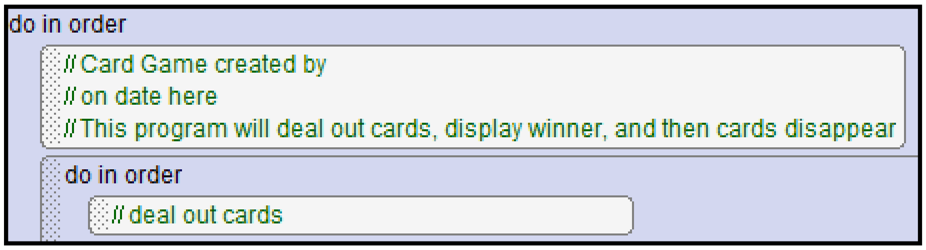


```
do in order
    // Card Game created by
    // on date here
    // This program will deal out cards, display winner, and then cards disappear
```

8. Our scene is complete, so we will move to the next task: "Player cards are dealt out onto screen." This will involve several steps so we will break this down into more detailed steps. This process is called stepwise refinement.

Player Cards are dealt out onto screen
- Make an announcement that cards will now be dealt out
- Playing card for player 1 moves from off screen to cone1 marker
- Playing card for player 2 moves from off screen to cone 2 marker

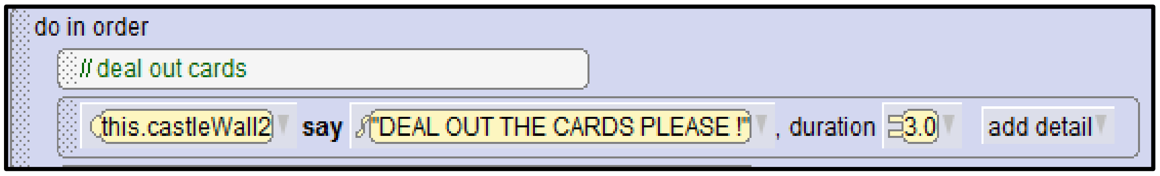
9. Grouping sections of program together will make the program more understandable. Therefore, we will begin a new section of code by dragging a DO IN ORDER to our program and then adding a comment to this section stating that this is the deal out cards section.



```
do in order
    // Card Game created by
    // on date here
    // This program will deal out cards, display winner, and then cards disappear
    do in order
        // deal out cards
```

10. To make the announcement that cards will now be dealt out, we will choose any object on the scene and choose the **say** procedure. The custom **TextString** should be **DEAL OUT THE CARDS PLEASE!**

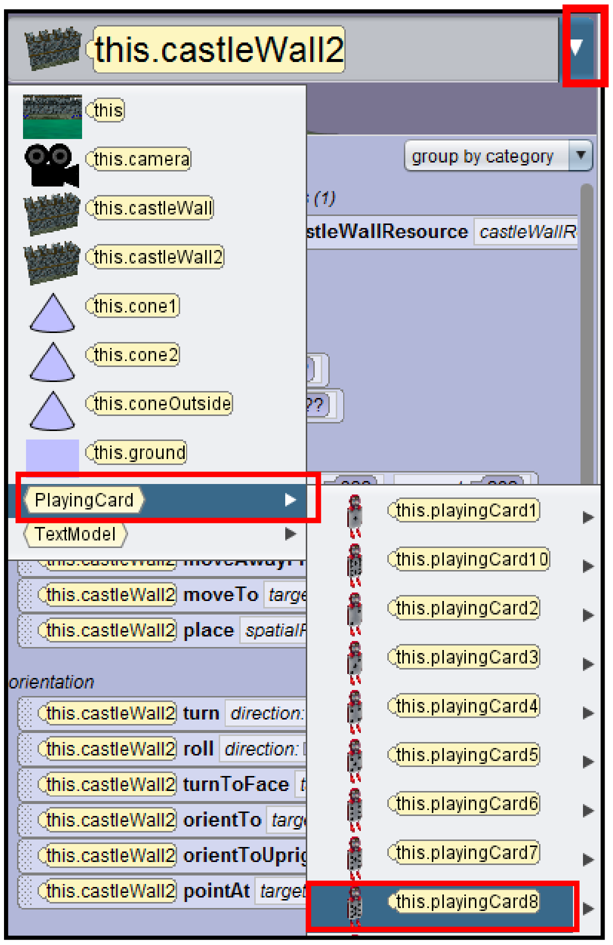
The following statement has the castleWall2 displaying the words. It also has the added detail of duration of 3 seconds.



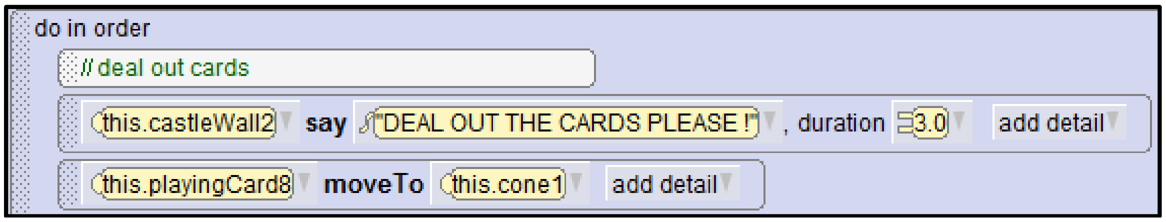
```
do in order
    // deal out cards
    this.castleWall2 say "DEAL OUT THE CARDS PLEASE !" , duration 3.0   add detail
```

11. Click on **Run** button in upper left corner to see the animation so far.

12. A playing card needs to be dealt out onto the screen. We have 10 cards so we will just arbitrarily pick to deal out card 8 to player1. It should move out to our invisible marker of cone1. To choose playingCard8, you will need to click on the down arrow of the object list, choose PlayingCard, and then choose **this.playingCard8** as follows:

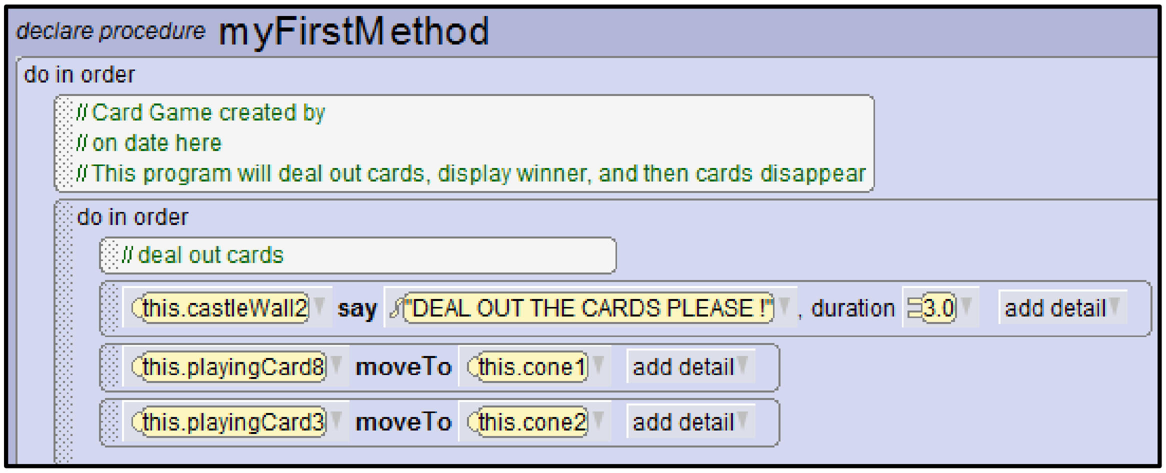


13. Once the **playingCard8** is selected. Choose the **procedure** of **moveTo** and drag it to the coding area. It will ask you what you are moving to and you should choose **cone1**. The statement should be placed in the inner DO IN ORDER block and look as follows:



```
do in order
    // deal out cards
    this.castleWall2 say "DEAL OUT THE CARDS PLEASE !" , duration 3.0 add detail
    this.playingCard8 moveTo this.cone1 add detail
```

14. A second card should be dealt out for player2. Arbitrarily, we will choose **playingCard3**. Select playingCard3 as your object and then choose the **moveTo** procedure. Choose to move to **cone2**. The program should look as follows:



```
declare procedure myFirstMethod
do in order
    // Card Game created by
    // on date here
    // This program will deal out cards, display winner, and then cards disappear
    do in order
        // deal out cards
        this.castleWall2 say "DEAL OUT THE CARDS PLEASE !" , duration 3.0 add detail
        this.playingCard8 moveTo this.cone1 add detail
        this.playingCard3 moveTo this.cone2 add detail
```

15. Click on the RUN button to see the animation play. If done correctly, an object should say “DEAL OUT THE CARDS PLEASE!”, then playingCard8 will appear on screen, and then playingCard3 will appear on screen. Make adjustments to your program if it isn’t working correctly.

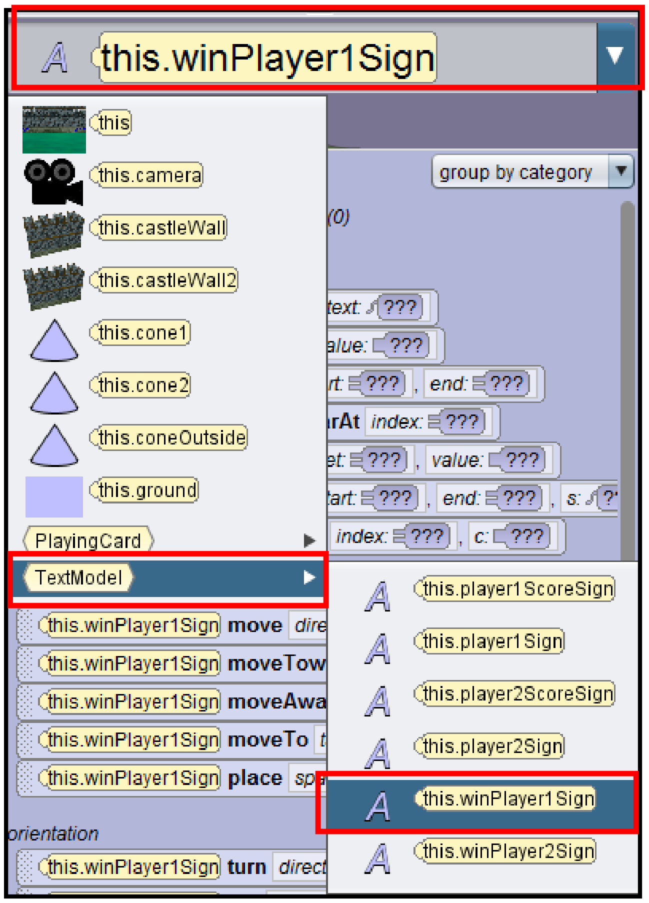
16. In the storyboard, the next task is “display winner” as seen below:

- Scene opens with the CardGame
- Player cards are dealt out onto screen
- Display winner
- Have these cards disappear
- New Player cards are dealt out on screen
- Deteremine and display winner
- Have these cards disappear

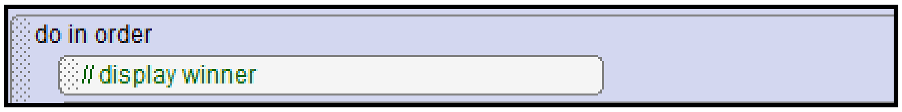
17. Let's break that down into the several steps it will take to display winner.

Display winner
• Flash the WIN sign for player 1
• Change the score for player 1

18. To flash the WIN sign for player 1 (named as winPlayer1Sign), we will have it show on screen (opacity of 1) and then disappear (opacity of 0) and do this process three times. Set the object to be the TextModel of **winPlayer1Sign** as follows:

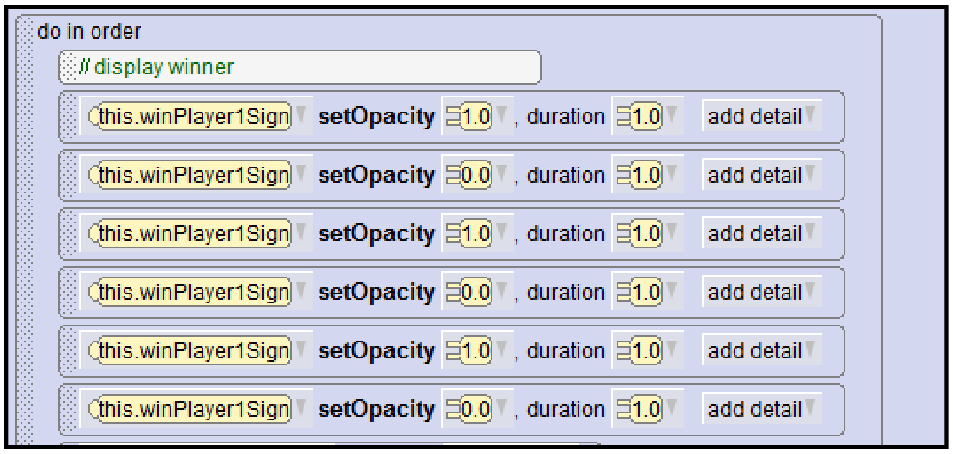


19. Drag a DO IN ORDER block to the program. Drag up a comment line and state that this section of program is to determine and display winner.



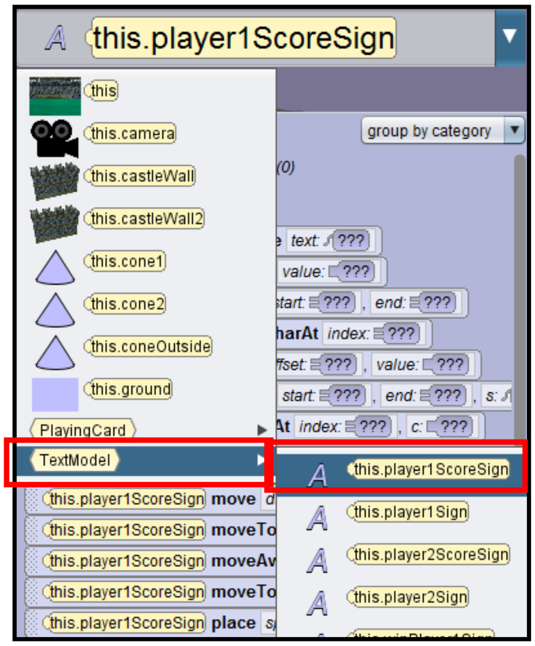
```
do in order
  // display winner
```

20. In the procedures, drag the **setOpacity** method to the editor. Set the **opacity** to be **1.0**. This makes the WIN sign display on the screen. To make it then disappear, you would drag the setOpacity method to the editor and set it to 0.0. (You can also copy these tiles by holding down the CTRL key and dragging the tile down to empty area). You should continue to have the opacity of this sign change between 1.0 and 0.0. This will cause a flashing effect. You may also change the duration of each to less than 2 seconds to make it flash quicker. In addition, you can add more copies of the statements to make it flash more often. The following shows this section of the code with a one second duration for each statement and flashing 3 times:

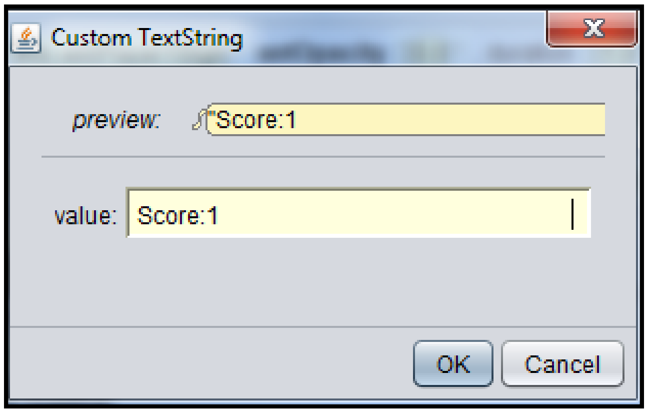


```
do in order
    // display winner
    this.winPlayer1Sign setOpacity 1.0 , duration 1.0 add detail
    this.winPlayer1Sign setOpacity 0.0 , duration 1.0 add detail
    this.winPlayer1Sign setOpacity 1.0 , duration 1.0 add detail
    this.winPlayer1Sign setOpacity 0.0 , duration 1.0 add detail
    this.winPlayer1Sign setOpacity 1.0 , duration 1.0 add detail
    this.winPlayer1Sign setOpacity 0.0 , duration 1.0 add detail
```

21. In this section, the score also needs to be changed. The player 1 score sign was named player1ScoreSign and is a TextModel. Set the object to be:



22. Once the player1ScoreSign is the selected object. Drag the **setValue** procedure to the program. It will ask what the value should be and choose **Custom TextString** and then type in **Score: 1** in the window as follows:

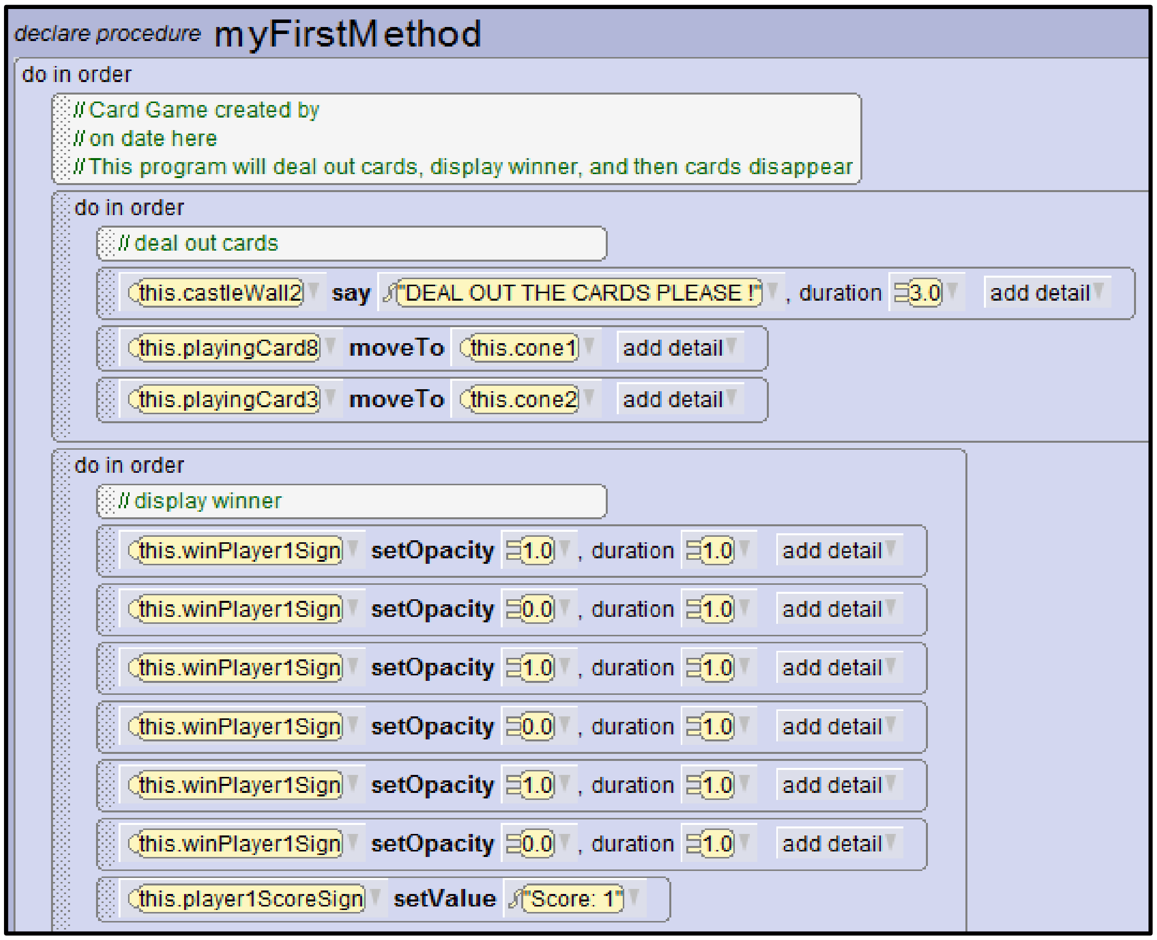


Custom TextString

preview: "Score:1

value: Score:1

23. The above will produce an Alice statement that will make the score appear to be 1. The entire program should look as follows:



```
declare procedure myFirstMethod
do in order
    // Card Game created by
    // on date here
    // This program will deal out cards, display winner, and then cards disappear
    do in order
        // deal out cards
        this.castleWall2 say "DEAL OUT THE CARDS PLEASE !" , duration 3.0 add detail
        this.playingCard8 moveTo this.cone1 add detail
        this.playingCard3 moveTo this.cone2 add detail
    do in order
        // display winner
        this.winPlayer1Sign setOpacity 1.0 , duration 1.0 add detail
        this.winPlayer1Sign setOpacity 0.0 , duration 1.0 add detail
        this.winPlayer1Sign setOpacity 1.0 , duration 1.0 add detail
        this.winPlayer1Sign setOpacity 0.0 , duration 1.0 add detail
        this.winPlayer1Sign setOpacity 1.0 , duration 1.0 add detail
        this.winPlayer1Sign setOpacity 0.0 , duration 1.0 add detail
        this.player1ScoreSign setValue "Score: 1"
```

24. Run the animation to see how well the program is working. The program should deal out the cards, the WIN sign should flash for player 1 and then the score for player 1 should change to 1. If your program is not working, you will need to review the above statements and make adjustments to make it work.

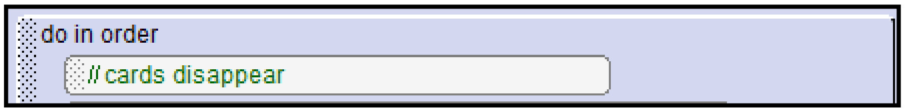
25. Reviewing the storyboard, we move on to the next task, which is "Have these cards disappear."

- Scene opens with the CardGame
- Player cards are dealt out onto screen
- Display winner
- Have these cards disappear
- New Player cards are dealt out on screen.
- Deteremine and display winner
- Have these cards disappear

26. We have to think which steps are involved in making the cards disappear. We would like the cards to spin and then move off the screen to their original location. Let's break this section down as:

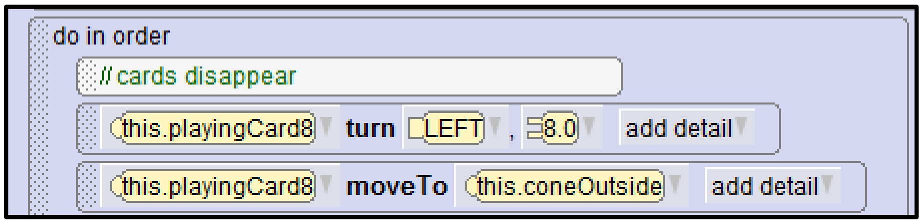
Have the cards disappear.
- Make the player1's card spin
- Make the player1's card return to stack of cards
- Make the player2's card spin
- Have the player2's card return to stack of cards

27. Let's start a new DO IN ORDER block and drag a comment tile up to the block that states that this section makes the cards disappear:



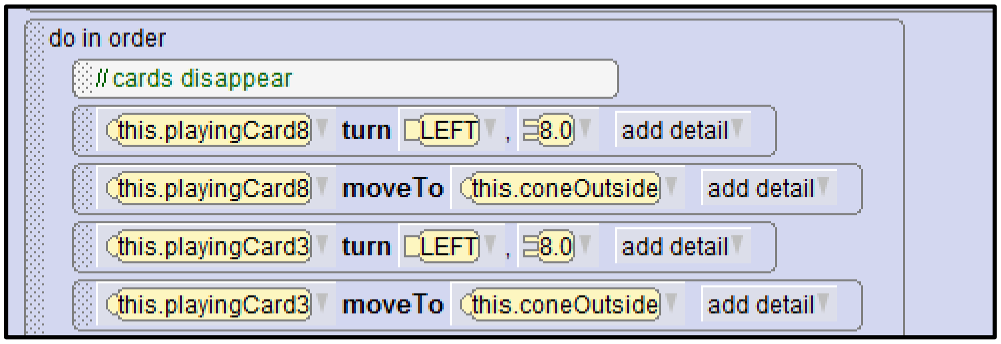
```
do in order
    // cards disappear
```

28. Set the object to be **playingCard8** since that is player1's card. Drag the **turn** procedure to the editor and choose for a **LEFT** with a rotation of **8.0**. Also, for this same object, drag the **moveTo** procedure to the editor and choose to have it move to **coneOutside**. These 2 statements will make the left card on screen spin around 8 times then move back to original stack of cards on left side. The statements should be:



```
do in order
    // cards disappear
    this.playingCard8 turn LEFT , 8.0 add detail
    this.playingCard8 moveTo this.coneOutside add detail
```

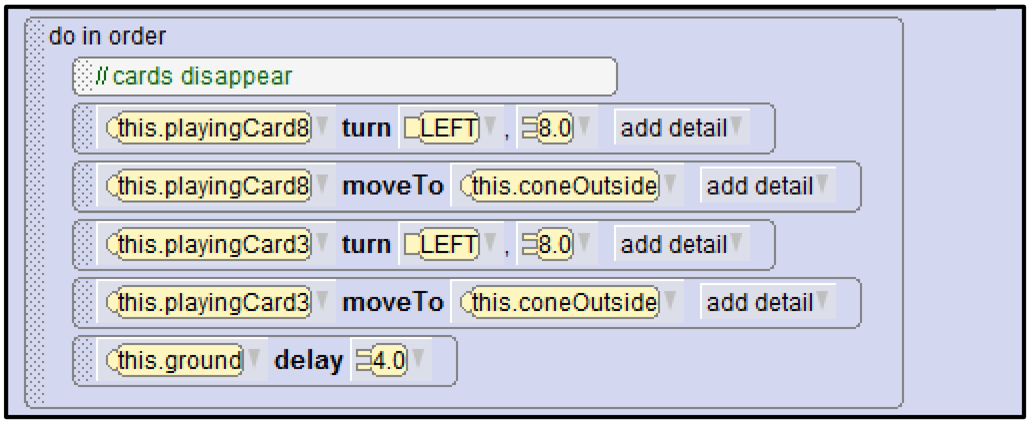
29. Run the program to see that everything is working. Change to **playerCard3** and drag the **turn** procedure to the program and make it turn 8 revolutions to the LEFT. Drag the **moveTo** procedure to the program and tell it to move to the coneOutside. You should now have the following:



```
do in order
    // cards disappear
    this.playingCard8 turn LEFT , 8.0 add detail
    this.playingCard8 moveTo this.coneOutside add detail
    this.playingCard3 turn LEFT , 8.0 add detail
    this.playingCard3 moveTo this.coneOutside add detail
```

30. Run the program and make sure it is working correctly. Both cards that were dealt out, should now disappear. They are returning to the stack of cards, but you won't be able to see that.

31. Since we will be adding more sections to continue this game, we would like to have a 4 second pause here in this section before it begins the game again. This is done with a delay. Change the object to the ground. You will see that there are very few procedures for the ground. One of them is a delay. Drag the **delay** procedure to the bottom of the DO IN ORDER for this section. When asked for a number, choose **Custom Decimal Number** and then choose **4** on the calculator and press OK. This means there will be a 4 second delay before we begin the game again.



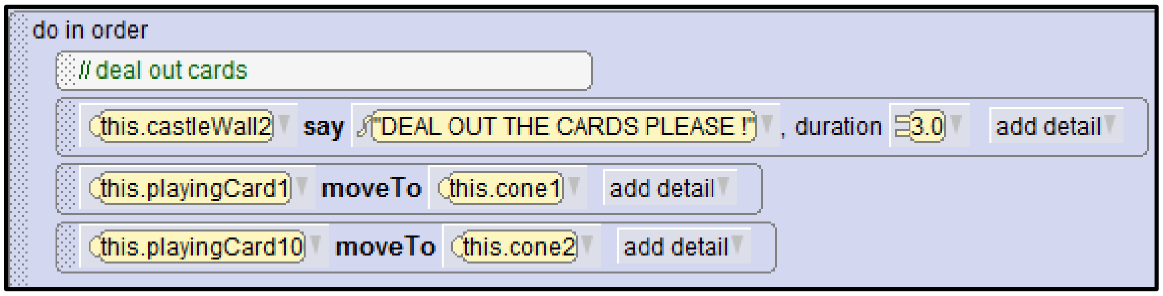
```
do in order
    // cards disappear
    this.playingCard8 turn LEFT , 8.0 add detail
    this.playingCard8 moveTo this.coneOutside add detail
    this.playingCard3 turn LEFT , 8.0 add detail
    this.playingCard3 moveTo this.coneOutside add detail
    this.ground delay 4.0
```

32. We would like to play the game again with player 2 winning. We will arbitrarily choose for playingCard1 to be dealt to player1 and playingCard10 to be dealt to player2 so that player2 has the higher card and will win. Let's review the original storyboard and see where we are:

- Scene opens with the CardGame
- Player cards are dealt out onto screen
- Display winner
- Have these cards disappear
- New Player cards are dealt out on screen
- Deteremine and display winner
- Have these cards disappear

33. We will duplicate previous sections of this program by copying/pasting them. In later chapters you will learn better ways of doing this, but for now the copy/paste will work fine. The first section to be duplicated will be the “deal out cards” section. Hold down the CTRL key *(alt/option key on the Mac)* and drag the entire “deal out cards” DO IN ORDER BLOCK to the Clipboard in upper right corner. If you don’t hold down the CTRL key, it will cut the block of code instead of copying it. If you hover over the clipboard, it will turn green when the copy/cut is successful. Then drag the clipboard to the bottom of the programming area. If everything is done correctly, you should have a deal out cards block at the beginning of the program and a copy of it at the bottom of the program.

34. The lines in the new block at bottom of program, need to be adjusted. Why? We no longer want to use playingCards 8 and 3. We want player 2 to win with playingCard10 and we want player 1 to have playingCard1. Adjust the lines in bottom section so that you click on arrow next to playingCard8 and change it to playingCard1. Adjust the line referring to playingCard3 to refer to playingCard10.



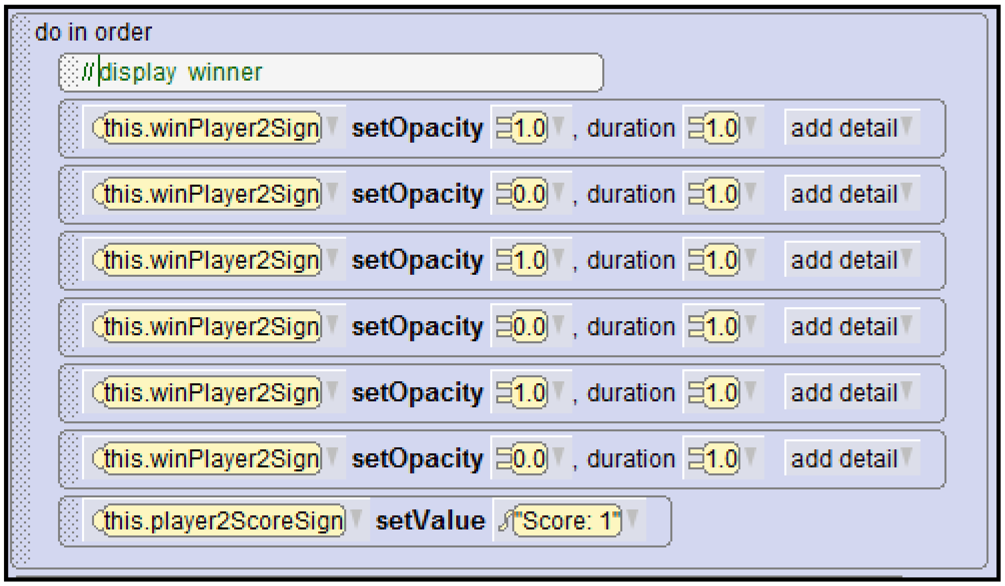
```
do in order
    // deal out cards
    this.castleWall2 say "DEAL OUT THE CARDS PLEASE !" , duration 3.0 add detail
    this.playingCard1 moveTo this.cone1 add detail
    this.playingCard10 moveTo this.cone2 add detail
```

35. Run the program to see if the 2 cards in the second game are dealt out correctly.

36. According to the storyboard, our next step is to “Display winner”. We have this already programmed into a block of code that we can copy and adjust. CTRL and drag the entire “display winner” DO IN ORDER BLOCK to the Clipboard. Then drag the copy of this section to the bottom of the editor.

37. This section will need adjusted. Look at the statements and try to decide what should change. Player1 has a 1 and Player2 has a 10. Player2 should win this time. You will need to adjust the setOpacity statements to refer to winPlayer2Sign. This is done by clicking the down arrow next to winPlayer1Sign and choosing winPlayer2Sign from the dropdown list.

38. The last statement of this section changes the score but without adjustment, it changes the score for Player1. We need to change the score for Player 2. Thus, in the last statement, adjust the player1ScoreSign, by clicking on the down arrow next to it, and choosing player2ScoreSign from the dropdown list. The entire new section should now appear as follows:

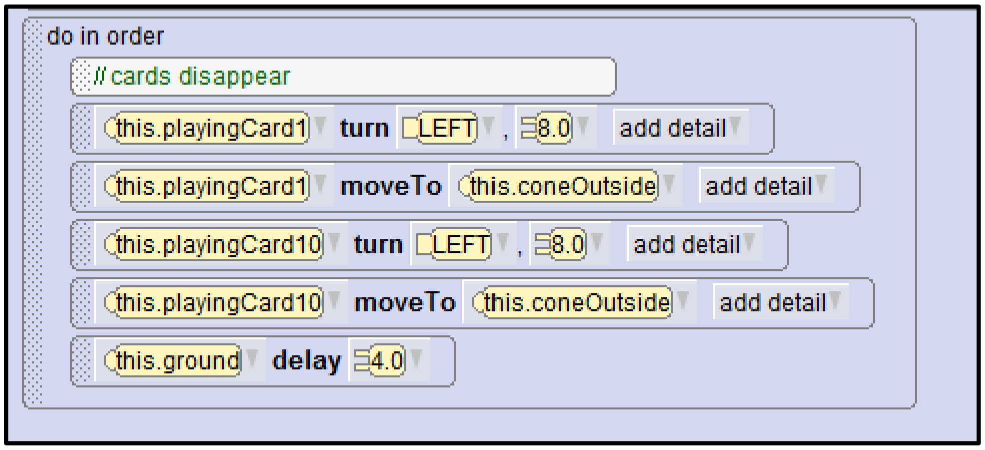


```
do in order
    //display winner
    this.winPlayer2Sign setOpacity 1.0 , duration 1.0 add detail
    this.winPlayer2Sign setOpacity 0.0 , duration 1.0 add detail
    this.winPlayer2Sign setOpacity 1.0 , duration 1.0 add detail
    this.winPlayer2Sign setOpacity 0.0 , duration 1.0 add detail
    this.winPlayer2Sign setOpacity 1.0 , duration 1.0 add detail
    this.winPlayer2Sign setOpacity 0.0 , duration 1.0 add detail
    this.player2ScoreSign setValue "Score: 1"
```

39. Run the program to see if the WIN sign flashes for player 2 in the second game. Did the score change for player 2 also?

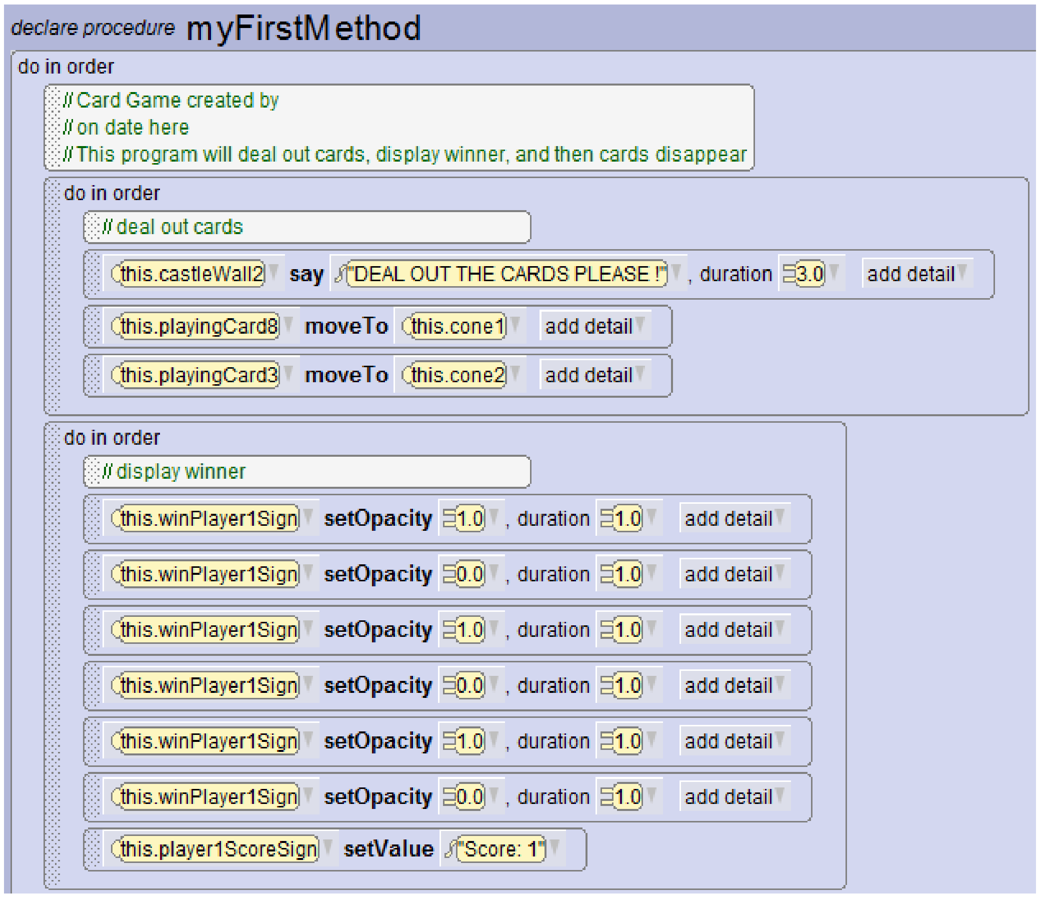
40. The last task of the storyboard is to have these last 2 cards disappear. Again, we have that code in an earlier section commented as "cards disappear". Copy that section to the clipboard by using the CTRL key. Drag the copy to the bottom of the editor.

41. What needs adjusted? THINK! These statements are referring to playingCard8 and playingCard3 because that is what they were in the first game. However, in this second game, the playing cards used were playingCard1 and playingCard10. Adjust those lines respectively so that you have the following:

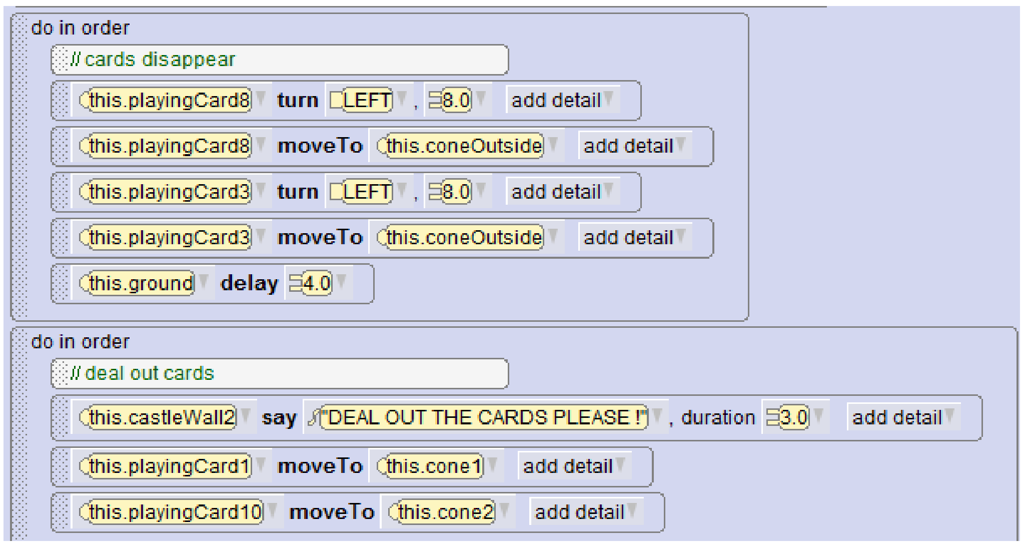


```
do in order
    // cards disappear
    this.playingCard1 turn LEFT , 8.0 add detail
    this.playingCard1 moveTo this.coneOutside add detail
    this.playingCard10 turn LEFT , 8.0 add detail
    this.playingCard10 moveTo this.coneOutside add detail
    this.ground delay 4.0
```

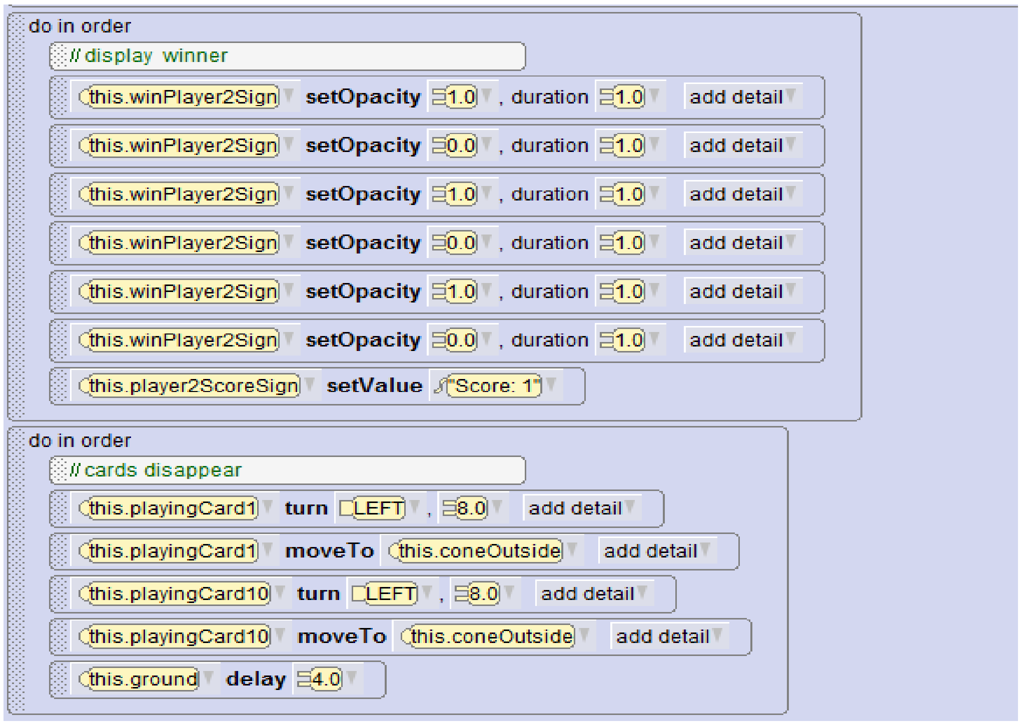
42. The complete program is as follows:



```
declare procedure myFirstMethod
do in order
    // Card Game created by
    // on date here
    // This program will deal out cards, display winner, and then cards disappear
    do in order
        // deal out cards
        this.castleWall2 say "DEAL OUT THE CARDS PLEASE !" , duration 3.0 add detail
        this.playingCard8 moveTo this.cone1 add detail
        this.playingCard3 moveTo this.cone2 add detail
    do in order
        // display winner
        this.winPlayer1Sign setOpacity 1.0 , duration 1.0 add detail
        this.winPlayer1Sign setOpacity 0.0 , duration 1.0 add detail
        this.winPlayer1Sign setOpacity 1.0 , duration 1.0 add detail
        this.winPlayer1Sign setOpacity 0.0 , duration 1.0 add detail
        this.winPlayer1Sign setOpacity 1.0 , duration 1.0 add detail
        this.winPlayer1Sign setOpacity 0.0 , duration 1.0 add detail
        this.player1ScoreSign setValue "Score: 1"
```



```
do in order
    // cards disappear
    this.playingCard8 turn LEFT , 8.0 add detail
    this.playingCard8 moveTo this.coneOutside add detail
    this.playingCard3 turn LEFT , 8.0 add detail
    this.playingCard3 moveTo this.coneOutside add detail
    this.ground delay 4.0
do in order
    // deal out cards
    this.castleWall2 say "DEAL OUT THE CARDS PLEASE !" , duration 3.0 add detail
    this.playingCard1 moveTo this.cone1 add detail
    this.playingCard10 moveTo this.cone2 add detail
```



```
do in order
    // display winner
    this.winPlayer2Sign setOpacity 1.0 , duration 1.0 add detail
    this.winPlayer2Sign setOpacity 0.0 , duration 1.0 add detail
    this.winPlayer2Sign setOpacity 1.0 , duration 1.0 add detail
    this.winPlayer2Sign setOpacity 0.0 , duration 1.0 add detail
    this.winPlayer2Sign setOpacity 1.0 , duration 1.0 add detail
    this.winPlayer2Sign setOpacity 0.0 , duration 1.0 add detail
    this.player2ScoreSign setValue "Score: 1"
do in order
    // cards disappear
    this.playingCard1 turn LEFT , 8.0 add detail
    this.playingCard1 moveTo this.coneOutside add detail
    this.playingCard10 turn LEFT , 8.0 add detail
    this.playingCard10 moveTo this.coneOutside add detail
    this.ground delay 4.0
```

43. This exercise allowed you to set up an animated card game with the 2 players. The player dealt the higher card wins. We executed the first game with player1 winning and second game with player2 winning. You should run this program and make sure everything is working.

44. Make sure to save this file as **CardGame** in **Chapter1Exercises** folder. You can practice coding by adding other elements to this program but make sure you save your practice with a different filename so that you have the completed version for future projects.

Summary

- Line by line instructions that tell a computer how to perform a task is called a **computer program**.
- A programming language that is written at the very low technical circuitry level of the computer is called a **low-level programming language**.
- **High-level programming languages** allow programmers to write programs using English terms.
- James Gosling at Sun Microsystems is credited with creating Java programming language. It was brought to public in 1995.
- Java is an object-oriented programming language that is platform-independent. The Java slogan is "You can write once and run anywhere." The compiler creates a bytecode file (with class extension) that can be used on all types of computer systems (MAC, Linux, Windows, etc.) Bytecode is then fed to a Java Virtual Machine (JVM) where they are interpreted and executed.
- Computer languages each have their own **syntax**, or rules of the language.
- When you write a program, indenting is important so that you and other humans can understand your program.
- In every program that you write, there should be comment lines at beginning of program. There should be a description of the program, author (you), date, and the JDK used for the program.
- Compilation error messages try to pinpoint the problem in your program but they are not always helpful. You may have to look around for the error or correct just the errors that you do understand and then compile again. You will get better at doing this and understanding this with additional experience.
- If your program is compiling fine, there still is no guarantee that it will work. An error-free compilation program only means that the Java compiler understands your commands, but the commands may not do what you want.
- The curly braces in Java are crucial. The symbols { } are generally found to the right of the "P" key.
- Programming can be very frustrating but it also can be rewarding when you succeed.

Review Questions

1. JDK stands for:
    a. Java Details Kit
    b. Java Development Kit
    c. Java Decoder Kit
    d. Java Debugger Kit

2. All programming languages work on the Internet.
    a. True
    b. False

3. Java and JavaScript are the same programming language.
    a. True
    b. False

4. There are many high-level programming languages for computers.
    a. True
    b. False

5. The rules of a programming language are its ___________.
    a. Vocabulary
    b. Syntax
    c. Logic
    d. Flowchart

6. Arguments to methods appear within
    a. Parentheses
    b. Semicolons
    c. Curly braces
    d. Quotation marks

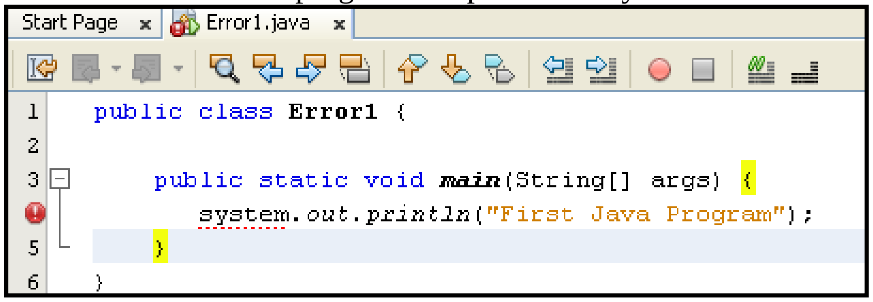
7. All Java application programs must have a method called _______.
    a. hello
    b. system
    c. main
    d. Java

8. Non-executing program statements that provide documentation to humans are called ____________.
    a. Notes
    b. Classes
    c. Commands
    d. Comments

9. Once a program has compiled without errors, it will always execute perfectly.
   a. True
   b. False

10. A computer ___________ tells a computer how to perform a task.
    a. Switch
    b. Program
    c. Interface
    d. Guide

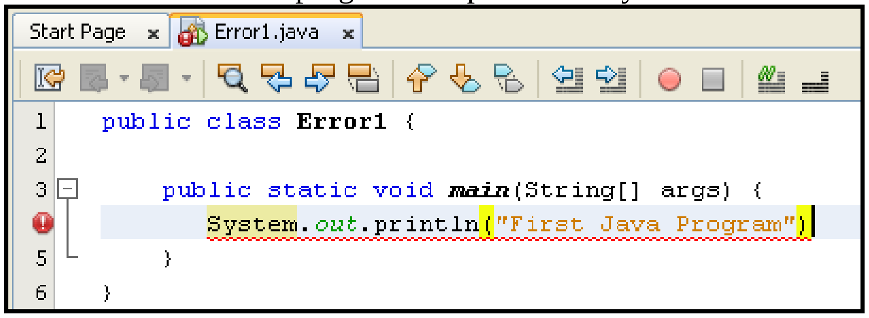
11. Look at the Java program at the top of the following illustration. This Java program was compiled and the compilation errors appear at bottom of screen. What needs to be corrected to make this program compile correctly?



```
1   public class Error1 {
2
3       public static void main(String[] args) {
            system.out.println("First Java Program");
5       }
6   }
```

    a. Change line 1 to say ***FirstJavaProgram*** instead of ***Error1***
    b. Change line 3 to say ***first*** instead of ***main***
    c. Change line 4 to say **System** instead of **system**
    d. Change line 4 to say "***Hello World***" instead of "***First Java Program***"

12. Look at the Java program at the top of the following illustration. This Java program was compiled and the Compilation errors appear at bottom of screen. What needs to be corrected to make this program compile correctly.



```
1   public class Error1 {
2
3       public static void main(String[] args) {
            System.out.println("First Java Program")
5       }
6   }
```

    a. Change line 1 to have semicolon at end of it.
    b. Change line 3 to have semicolon at end of it.
    c. Change line 4 to have semicolon at end of it.
    d. Change line 5 to have semicolon at end of it.

Solutions: *1) b 2) b 3) b 4) a 5) b 6) a 7) c 8) d 9) b 10) b 11) c 12) c*

Assignments

1-1 Cola Commercial: Your goal is to create a cola commercial using Alice and NetBeans.

- ***Analyze and understand the problem to be solved.*** We would like to create a cola commercial animation that is at least 5 seconds long. We need to take a look at the gallery to see what objects we have that could be used in our commercial. Then we need to set up the scene. You may have created the scene in chapter 0.
- ***Develop the logic to solve the program.*** We should develop a storyboard for our animation. The storyboard is a short description of what you want to happen in your animation.
- ***Code the solution in a programming language.*** Write the code in Alice. Give the file an appropriate name. Add your name, date, and a description of the program to the top of the program as comments.
- ***Test the program.*** Test the code in Alice.

1-2 Greeting Card: Your goal is to create an animated greeting card using Alice and NetBeans.

- ***Analyze and understand the problem to be solved.*** We would like to create a greeting card animation that is at least 5 seconds long. We need to take a look at the gallery to see what objects we have that could be used in our commercial. Then we need to set up the scene. You may have created the scene in chapter 0.
- ***Develop the logic to solve the program.*** We should develop a storyboard for our animation. The storyboard is a short description of what you want to happen in your animation.
- ***Code the solution in a programming language.*** Write the code in Alice. Give the file an appropriate name. Add your name, date, and a description of the program to the top of the program as comments.
- ***Test the program.*** Test the code in Alice.

1-3 Animation: Your goal is to create a short animation using Alice and NetBeans.

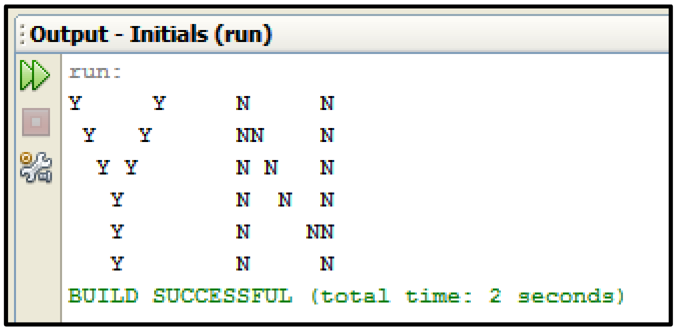
- ***Analyze and understand the problem to be solved.*** We would like to create a short animation of our choosing that is at least 5 seconds long. We need to take a look at the gallery to see what objects we have that could be used in our commercial. Then we need to set up the scene. You may have created the scene in chapter 0.
- ***Develop the logic to solve the program.*** We should develop a storyboard for our animation. The storyboard is a short description of what you want to happen in your animation.
- ***Code the solution in a programming language.*** Write the code in Alice. Give the file an appropriate name. Add your name, date, and a description of the program to the top of the program as comments.
- ***Test the program.*** Test the code in Alice.

1-4 Card Game adjusted: Your goal is to change the CardGame created in exercise 5 of this chapter to have the playing cards dance out onto the screen by using some hand movements or some leg movements.

- ***Analyze and understand the problem to be solved.*** We would like to create a short animation to have the playing cards dance out onto the screen by using some hand movements or some leg movements. We already have the playing card objects in the scene. Only the "deal Out Cards" code section should be changed.
- ***Develop the logic to solve the program.*** We should develop a storyboard for our animation. The storyboard is a short description of what you want to happen in your animation.
- ***Code the solution in a programming language.*** Write the code in Alice. Save this file as **MyCardGame**. Add your name, date, and a description of the program to the top of the program as comments.
- ***Test the program.*** Test the code in Alice.

1-5 Printing Initials: Your goal is to print your initials in block letters. In this exercise, instead of just keying in a program, we will learn the entire programming process to solve a problem. Programming is more than just keying in computer programs and testing them. Most programmers must go through a process to logically solve a problem or obtain a goal. The steps involved in computer programming are:

- ***Analyze and understand the problem to be solved.*** In this case, we would like the computer to display your initials in block letters. For example, if your initials were "YN" it would look as follows:



```
run:
Y     Y     N     N
 Y   Y      NN    N
  Y Y       N N   N
   Y        N  N  N
   Y        N    NN
   Y        N     N
BUILD SUCCESSFUL (total time: 2 seconds)
```

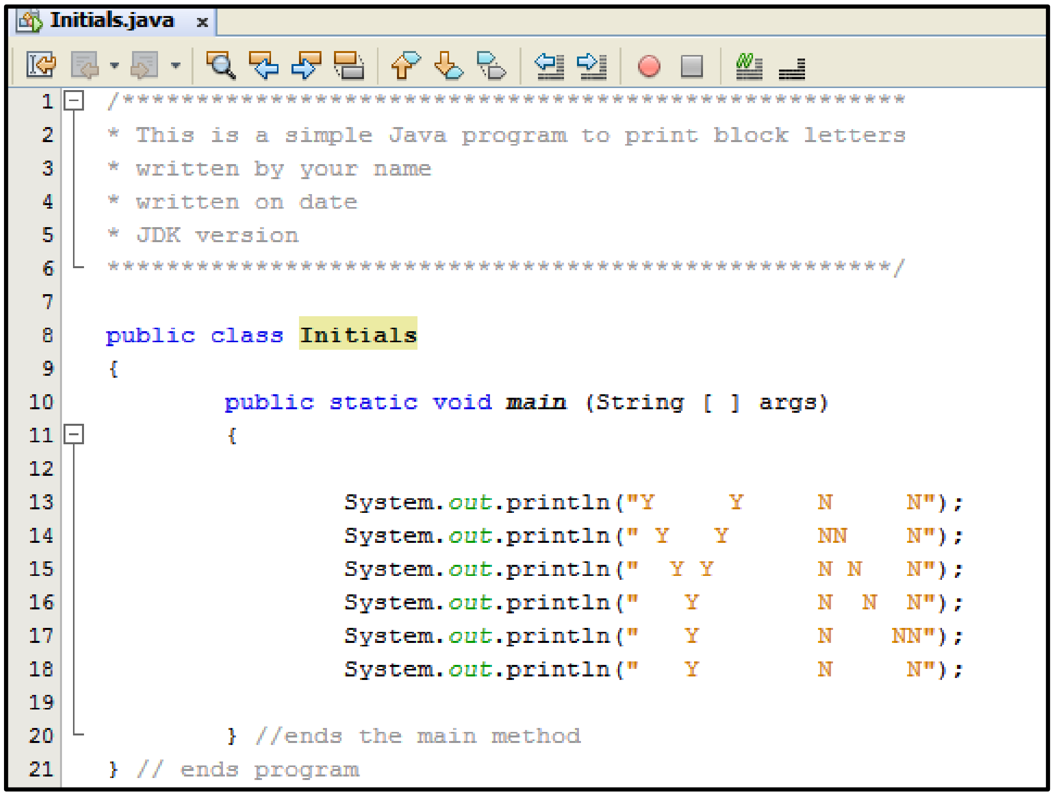
- ***Develop the logic to solve the program.*** We know how to print characters on a line and the above looks like to could consist of several print statements. We can graph the block letters on graph paper, so we can see the spacing. The letters should be about 7 by 7 characters wide and have 5 spaces in between letters.

  Below is a blank grid so that you can draw your initials.

  Sample solution for initials YN (your name). You should not use YN unless your initials are YN.

Y						**Y**						**N**						**N**
	Y				**Y**							**N**	**N**					**N**
		Y		**Y**								**N**		**N**				**N**
			Y									**N**			**N**			**N**
			Y									**N**				**N**		**N**
			Y									**N**					**N**	**N**
			Y									**N**						**N**

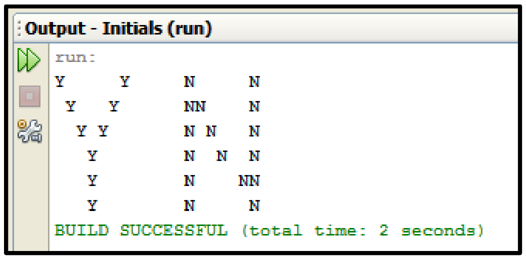
- ***Code the solution in a programming language.*** We will need to code 7 print statements to accomplish the above graphic of the “YN” block letters. The spacing in these *System.out.println* statements must be perfect. Here is the Java code.



```java
/*****************************************************
 * This is a simple Java program to print block letters
 * written by your name
 * written on date
 * JDK version
 *****************************************************/

public class Initials
{
        public static void main (String [ ] args)
        {

                System.out.println("Y     Y     N     N");
                System.out.println(" Y   Y      NN    N");
                System.out.println("  Y Y       N N   N");
                System.out.println("   Y        N  N  N");
                System.out.println("   Y        N    NN");
                System.out.println("   Y        N     N");

        } //ends the main method
} // ends program
```

Enter the coded solution into a new project in NetBeans. Name this project **Initials**.

- ***Test the program.*** Translate the program into the language understood by the computer. The Java compiler will do this. When you run your program, it is compiled and a bytecode class file is created if there aren’t any syntax errors. This bytecode file may be used on any operating system. Once you have a bytecode file, you will see the results of your program. If you have syntax errors, you will need to fix these errors before running your program (red underlines throughout your code).



```
run:
Y     Y     N     N
 Y   Y      NN    N
  Y Y       N N   N
   Y        N  N  N
   Y        N    NN
   Y        N     N
BUILD SUCCESSFUL (total time: 2 seconds)
```

Now, it is your turn to go through the programming process above and create block letters of YOUR initials.