CSUS COLLEGE OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science

CSc 133 – Object-Oriented Computer Graphics Programming
Summer 2019 – Dr. Doan Nguyen (Credits to Professor Clevenger).

Assignment #1: Class Associations & Interfaces

Introduction

In this summer 2019 session, we will be studying object-oriented computer graphics programming by developing a program which is a variation of a classic arcade and home video game called "Asteroids" (Version 6 – 5/23/19). In this game, you will be controlling a space ship flying through a field of asteroids, firing missiles to destroy asteroids which threaten to collide with and destroy your ship. Occasionally, enemy ships attempt to impede progress. Various other graphical operations will also be implemented to extend your game beyond the original version.

If you have never seen the original Asteroids game, you can see a sample of the (Arcade) Asteroids game in Youtube. https://www.youtube.com/watch?v=WYSupJ5r2zo. However, your game will be slightly different, and it is not necessary to be familiar with the original Asteroids game to do any of the assignments during the semester.

The initial version of your game will be *text-based*. As the session progresses we will add graphics, animation, and sound. The goal of this first assignment is to develop a good initial class hierarchy and control structure, and to implement it in CN1/Java. This version uses keyboard input commands to control and display the contents of a "game world" containing a set of objects in the game. In future assignments, many of the keyboard commands will be replaced by interactive GUI operations or be replaced with animation functions, but for now we will simply simulate the game in "text mode" with user input coming from the keyboard and "output" being lines of text on the screen.

Program Structure

Because you will be working on the same project all semester, it is extremely important to organize it correctly from the beginning. Pay careful attention to the class structure described below and make sure your program follows this structure accurately.

The primary class encapsulates the notion of a ***Game***. A game in turn contains several components: (1) a ***GameWorld*** which holds a collection of *game objects* and other state variables, and (2) a **play**() method to accept and execute user commands. Later, we will learn that a component such as GameWorld that holds the program's data is often called a **model**.

The top-level *Game* class also encapsulates the *flow of control* in the game (such a class is therefore sometimes called a **controller**). The controller enforces rules such as what actions a player may take and what happens as a result. This class accepts input in the form of keyboard commands from the human player and invokes appropriate methods in the game world to perform the requested commands – that is, to manipulate data in the game model.

In this first version of the program the top-level game class will also be responsible for displaying information about the state of the game. In future assignments, we will learn about a separate kind of component called a **View** which will assume that responsibility.

When you create your CN1 project, you should name the main class as **Starter**. Then you should modify the start() method of the Starter class so that it would construct an instance

of the Game class. The other methods in Starter (i.e., init(), stop(), destroy()) should not be altered or deleted. The Game class must extend from the build-in Form class (which lives in **com.codename1.ui** package). The Game constructor instantiates a GameWorld, calls a GameWorld method **init**() to set the initial state of the game, and then starts the game by calling a Game method **play**(). The play() method then accepts keyboard commands from the player and invokes appropriate methods in GameWorld to manipulate and display the data and game state values in the game model. Since CN1 does not support getting keyboard input from command prompt (i.e., the standard input stream, System.in, supported in Java is not available in CN1) the commands will be entered via a text field added to the form (the Game class). Refer to "Appendix – CN1 Notes" for the code that accepts keyboard commands through the text field located on the form.

The following shows the pseudo-code implied by the above description. It is important for things that we will do later that your program follows this organization:

```
class Starter {
//other methods
    public void start() {
        if(current != null){
            current.show();
            return;
        }
        new Game();
    }
//other methods
}
```

```
public class GameWorld {
   public void init(){
      //code here to create the
      //initial game objects/setup
   }
   // additional methods here to
   // manipulate world objects and
   // related game state data
}
```

```
import com.codename1.ui.Form;
public class Game extends Form{
   private GameWorld gw;

   public Game() {
      gw = new GameWorld();
      gw.init();
      play();
   }

   private void play() {
      // code here to accept and
      // execute user commands that
      // operate on the game world
      //(refer to "Appendix - CN1
      //Notes" for accepting
      //keyboard commands via a text
      //field located on the form)
   }
}
```

Game World Objects

For now, assume that the game world size is fixed and covers 1024(width) x 768(height) area (although we are going to change this later). The origin of the "world" (location (0,0)) is the lower left hand corner. The game world contains a collection which aggregates objects of abstract type **GameObject**. There are two kinds of abstract game objects: "fixed objects" with fixed locations (which are fixed in place) and "moveable objects" with changeable locations (which can move or be moved about the world). For this first version of the game there is just a single kind of fixed object: a ***space station***; and there are three kinds of moveable objects: ***ships (Player Ship (PS) and Non-Player Ship (NPS))***, ***missiles*** (which are objects fired by ships), and ***asteroids***. Later we will see that there are other kinds of game objects as well.

The various game objects have attributes (fields) and behaviors ("functions" or "methods") as defined below. These definitions are requirements which must be properly implemented in your program.

- All game objects have a *location*, defined by floating point values X and Y. (you can use float or double to represent it) non-negative values X and Y which initially should be in the range 0.0 to 1024.0 and 0.0 to 768.0, respectively. The point (X,Y) is the center of the object. Hence, initial locations of all game objects should always be set to values such that the objects' centers are contained in the world. All game objects provide the ability for external code to obtain their location. By default, game objects provide the ability to have their location changed, unless it is explicitly stated that a certain type of game object has a location which cannot be changed once it is created. Except for a player ***ship,*** initial locations of all the game objects should be assigned randomly when created.
  - Note: For future usages (i.e. specifying a point in a space), you might consider encapsulating a pair of coordinate values (X,Y) using a CN1's built-in classes Point or Point2D.
    See: www.codenameone.com/javadoc/com/codename1/ui/geom/Point.html
    www.codenameone.com/javadoc/com/codename1/ui/geom/Point2D.html

- All game objects have a color, defined by a int value (use static rgb() method of CN1's built-in **ColorUtil** class to generate colors). All objects of the same class have the same color (chosen by you), assigned when the object is created (e.g, ships could be green, missiles could be red, asteroids could be blue). All game objects provide the ability for external code to obtain their color. By default, game objects provide the ability to have their color changed, unless it is explicitly stated that a certain type of game object has a color which cannot be changed once it is created.

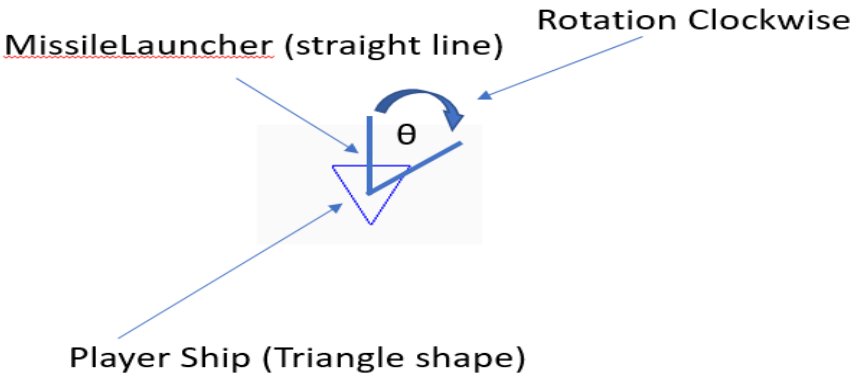
- Moveable objects have attributes *speed* and *direction (heading)*, which are used to define how they move through the world when told to do so. *Speed* is an integer in the range ***0-20***, initialized to a random value in that range unless otherwise stated. When a ship reaches maximum speed of 20, its firing missile can be increased with a small delta value. *Direction* is an integer in the range 0..359, initialized to a **random value** in that range unless specified otherwise, indicating a compass heading. (Thus, a direction of zero specifies heading "north" or "up the screen"; a direction of 90 specifies heading "east" or toward the right side of the screen, etc.)

- Moveable objects provide an interface that allows other objects to tell them to move. Telling a moveable object to *move()* causes the object to update its location based on its current speed and direction (heading). See below for details on updating a movable object's position when its *move()* method is invoked.

- Some moveable objects are *steerable*, meaning that they provide an interface that allows other objects to tell them to change their directions.

- *Player Ship (PS) is* steerable objects whose speed and direction (heading) can be controlled by a player. There is only *one player* ship in the game at any one time, and its initial location should be the center of the world (e.g, 512,384), with a speed of zero and a heading of zero (north). PS has an attribute called *missile count*, which is an integer representing the number of missiles the ship has available to be fired. PS starts with the maximum number of missiles, which is 10. Each time a ship fires a missile its missile count is decremented by one; when missile count reaches zero it can no longer fire any missiles. However, ships that fly to a space station automatically have their missile count reloaded to the original maximum number.

- A PS owns a MissileLauncher. It is a steerable object whose speed and location is the same as a PS. However, a direction of a MissileLauncher can be different from a PS. There is only one MissileLauncher per PS. Its initial location should be at the center of the world with speed of zero and heading of north. Graphically, in future Assignment 3 (a3) – not in this assignment, a MissleLauncher can be appeared like this when initialized:



Both classes, PlayerShip and MissileLauncher <u>are required to have a **composition** relation.</u>

- Non-Player Ship(s) (NPS) is NOT steerable objects. *NPS(s) are* movable objects that fly through space at a fixed speed and direction. *NPS(s)* also have a fixed attribute *size (small 15 or large 25),* an integer giving the dimensions of the NPS(s). The size (small or large), speed, and direction of a NPS are to be randomly generated when it is created. If a player's missile collides with a NPS, the NPS (and the missile) are destroyed and removed from the game, and the user scores some points (you may decide how many). If a *PS* collides with a NPS, the PS and the NPS are destroyed and the player loses one "life". A complete game consists of playing until three PS have been destroyed. A NPS also has a MissileLauncher. However, it is not steerable. It has a same direction a NPS. A NPS can randomly fires missile at PS (*a maximum of 4*). If a NPS's missile collides with a PS, the player loses one life and a missile is removed from the game.

- *Missiles* are moveable objects that are fired from ships's MissileLauncher. When a missile is fired, it has a direction matching its ship's MissileLauncher (that is, missiles are fired out the front of a launcher), and a speed that is some fixed amount greater than the speed of its ship (you may decide the amount). Missiles also have an attribute called *fuel level*, a positive integer representing the number of time units left before the missile becomes dead and disappears. A missile's fuel level gets decremented as time passes, and when it reaches zero the missile is removed from the game. All missiles have an initial fuel level of *15*. Note that while missiles are *moveable*, they are not *steerable*; that is, there is no way to change a missile's speed or heading once it is fired. The program must enforce this constraint.

- *Asteroids* are movable objects that tumble through space at a fixed speed and direction. Asteroids also have a fixed attribute *size (range 6..30 – Note: We will relax this assumption in later assignment),* an integer giving the dimensions of the asteroid. The size, speed, and direction of an asteroid are to be randomly generated when the asteroid is created. If a PS's missile collides with an asteroid, the asteroid (and the missile) are

destroyed and removed from the game, and the user scores some points (you may decide how many). If a player *ship* collides with an asteroid, the ship and the asteroid are destroyed and the player loses one "life". A complete game consists of playing until three player ships have been destroyed.

- Fixed objects have unique integer *identification (id) numbers*; every fixed object (of any subtype) has an integer id which is different from that of every other fixed object. (Note that this is an appropriate place to use a *static int* in a Java class; such a value can be used to keep track of the next available id number each time an instance of the class is constructed.) The program must enforce the rule that it is not possible to change either the ID number or the location of a fixed object once it has been created.

- *Space stations* are fixed objects that have an attribute called their *blink rate*; each space station blinks on and off at this rate (blink rate specifies the *period*; a blink rate of *6*, for example, means the light on the space station is on for six seconds and then off for six seconds). Blink rate is initialized to a random value between zero and six. Space stations also house an unlimited supply of missiles; any ship arriving at a space station automatically gets a full resupply of missiles (up to the maximum number of missiles a ship can hold)

The preceding paragraphs imply several *associations* between classes: an inheritance hierarchy, interfaces such as for *steerable* and *moving* objects, and aggregation associations between objects and where they are held. **You must first develop a UML diagram for the relationships, and then implement it in a Java/CN1 program**. Appropriate use of encapsulation, inheritance, aggregation, abstract classes, and interfaces are important grading criteria.

You must use a tool to draw your UML (e.g., Violet or any other UML drawing tool) and output your UML as a **pdf file** (e.g., print your UML to a pdf file). Your UML must show all important associations between your entities and utilize correct graphical notations. For your entities, you must use three-box notation (as mentioned in lecture) and show all the important fields and methods. You must indicate the visibility modifiers of your fields and methods, but you are not required to show parameters in methods and return types of the methods.

Game Play

When your program begins, there are no objects in the world. The player can enter commands (see below) to control various aspects of the game such as adding objects, moving the ship, firing missiles, etc. The game keeps track of time using a clock, which is simply an integer counter. The objective is to achieve the highest possible score in the minimum amount of time – that is, to destroy as many asteroids and NPS(s) as possible, flying to a space station to restock missiles as necessary, without losing ships, all in minimum time.

Commands

Once the game world has been created and initialized, the Game constructor is to call a method name play() to actually begin the game.play() accepts single-character commands from the player via the text field located on the form (the Game class) as indicated in the "Appendix – C1 Notes".

Any undefined or illegal input should generate an appropriate error message on the console and ignore the input. Single keystrokes invoke action -- the human hits "enter" after typing each command. **Additionally, any input condition where a function cannot be carried out, your program must output an error text. For example, user inputs a 'd' (decrease a**

speed of a player ship) command where a 's' command (add a player ship to the world) was not initiated before. The allowable input commands and their meanings are defined below (note that commands are case sensitive):

'**a**' – add a new **a**steroid to the world.

'**y**' – add a NPS to the world.

'**b**' – add a new **b**linking space station to the world.

'**s**' – add a PS to the world.

'**i**' – **i**ncrease the speed of the PS by a small amount.

'**d**' – **d**ecrease the speed of the PS by a small amount, as long as doing so doesn't make the speed go negative (the ship can't go backwards).

'**l**' (ell) – turn PS **l**eft by a small amount (change the heading of the ship by a small amount in the counter-clockwise direction). Note that determination of the actual new heading is not the responsibility of the game class; it simply instructs the game world that a change is to be made.

'**r**' – turn PS **r**ight (change the ship heading by a small amount in the clockwise direction).

'**>**' - a user controls the direction of the PS's MissileLauncher by revolving it about the center of the player ship in a clockwise direction. This command turns the MissileLauncher by small θ angle. You decide a value.

'**f**' – **f**ire a missile out the front of the PS. If the ship has no more missiles, an error message should be printed; otherwise, add to the world a new missile with a location, speed, and launcher's heading determined by the ship.

'**L**' – **L**aunch a missile out the front of the NPS. If the ship has no more missiles, an error message should be printed; otherwise, add to the world a new missile with a location, speed, and launcher's heading determined by the ship.

'**j**' – **j**ump through hyperspace. This command causes the ship to instantly jump to the initial default position in the middle of the screen, regardless of its current position. (This makes it easy to regain control of the ship after it has left visible space; later we will see a different mechanism for viewing the ship when it is outside screen space.)

'**n**' – load a **n**ew supply of missiles into the PS. This increases the ship's missile supply to the maximum. It is permissible to reload missiles before all missiles have been fired, but it is not allowed for a ship to carry more than the maximum number of missiles. Note that normally the ship would actually have to fly to a space station to reload; for this version of the game we will assume the station has transporters that can transfer new missiles to the ship regardless of its location.

'**k**' – a PS's missile has struck and **k**illed an asteroid; tell the game world to remove a missile and an asteroid and to increment the player's score by some amount of your choosing. You may choose any missile and asteroid to be removed; later we'll worry about missiles needing to be "close to" their victims.

'**e**' – a PS's missile has struck and **e**liminated a NPS; tell the game world to remove a missile and a NPS and to increment the player's score by some amount of your choosing. You

may choose any missile and any NPS to be removed; later we'll worry about missiles needing to be "close to" their victims.

'**E**' – a NPS's missile has struck and **E**xploded a PS; tell the game world to remove a missile and a PS. You may choose any missile and to decrement the count of lives left; later we'll worry about missiles needing to be "close to" their victims.

'**c**' – the PS has **c**rashed into an asteroid; tell the game world to remove the ship and an asteroid and to decrement the count of lives left; if no lives are left then the game is over. You may choose any asteroid to be removed; later we'll worry about asteroids needing to be "close to" the ship.

'**h**' – the PS has **h**it a NPS; tell the game world to remove the NPS and to decrement the count of lives left; if no lives are left then the game is over. You may choose any NPS to be removed; later we'll worry about NPS needing to be "close to" the PS.

'**x**' -- two asteroids have collided with and e**x**terminated each other; tell the game world to remove two asteroids from the game. You may choose any two asteroids to be removed; later we'll worry about the asteroids needing to be "close to" each other. You may ignore collisions between NPSs.

'**I**' -- one asteroid have collided and **i**mpacted the NPS; tell the game world to remove them from the game. You may choose one asteroid and one NPS to be removed; later we'll worry about the asteroid needing to be "close to" NPS.

'**t**' – tell the game world that the "game clock" has **t**icked. Each tick of the game clock has the following effects: (1) all moveable objects are told to update their positions according to their current direction and speed; (2) every missile's fuel level is reduced by one and any missiles which are now out of fuel are removed from the game; (3) each space station toggles its blinking light if the tick count modulo the station's blink rate is zero; and (4) the "elapsed game time" is incremented by one.

'**p**' – **p**rint a display (output lines of text) giving the current game state values, including: (1) current score (number of points the player has earned), (2) number of missiles currently in the ship, (3) current elapsed time, and (4) Number of lives. Output should be well labeled in easy-to-read format.

'**m**' – print a "**m**ap" showing the current world state (see below).

'**q**'– **q**uit, by calling the method `System.exit(0)` to terminate the program. Your program should first confirm the user's intent to quit before exiting.

Additional Details

- Each command must be encapsulated in a separate method in the Game class (later we will move those methods to other locations). Most of the game class command actions also invoke related actions in *GameWorld*. When the *Game* gets a command, it invokes one or more methods in the *GameWorld*, rather than itself manipulating game world objects.
- All classes must be designed and implemented following the guidelines discussed in class:
  - *All data fields must be private,*
  - *Accessors / mutators must be provided, but only where the design requires them,*
  - *Game world objects may only be manipulated by calling methods in the game world. It is never appropriate for the controller to directly manipulate game world data attributes.*

Moving objects need to determine their new location when their *move()* method is invoked, at each time tick ('**t**' command). The new location can be computed as follows:

`newLocation(x,y) = oldLocation(x,y) + (deltaX, deltaY),` *where*
`deltaX = cos(θ)*speed,`
`deltaY = sin(θ)*speed,` *and*
`θ = 90 – heading.`

In this assignment, we are assuming "time" is fixed at one unit per "tick", so "elapsed time" is 1. When computing for cos and sin function, using Math.cos(), Math.sin()), please expect the angles to be provided in radians not degrees. Please be sure to convert into radians.

- For this assignment, all output will be in text on the console; no "graphical" output is required. The "map" (generated by the '**m**' command) will simply be a set of lines describing the objects currently in the world, similar to the following:

```
Player Ship: loc=130.0,565.5 color=[0,255,0] speed=8 dir=90 missiles=5 Missile launcher dir = 50
PS's Missile: loc=180.3,100.0 color=[255,0,0] speed=10 dir=112 fuel=7
NPS's Missile: loc=50.9,540.2 color=[255,0,0] speed=5 dir=313 fuel=1
Non-Player Ship: loc=640.2,20.0 color=[128,128,128] speed=6 dir=20 size=10
Asteroid: loc=440.2,20.0 color=[128,128,128] speed=6 dir=0 size=6
Asteroid: loc=660.0,50.0 color=[128,128,128] speed=1 dir=90 size=26
Station: loc=210.2,800.0 color=[255,255,0] rate=4
```

Please be conformed to the format specified in above example. See **Appendix CN1 notes**, page 11 and 12, for more information. The grader will be enforcing this for grading consistency between students.

Note that the appropriate mechanism for implementing this output is to override the **`toString()`** method in each concrete game object class so that it returns a String describing itself. See the coding notes Appendix (see page 11 below) for additional details, including how to display single decimal value.

- Missiles are considered to have special radar that allows them to avoid hitting their own ships, other missiles, and space stations, so we will not worry about handling these collision conditions. Likewise, we will assume that space stations automatically avoid asteroids and NPS.

- The program must handle any situation where the player enters an illegal command – for example, a command to increase the ship speed when no ship has yet been added to the world – by printing an appropriate condition-specific error message on the console (for example, "Cannot execute 'increase' – no ship has been created") and otherwise simply waiting for a valid command.

- The program is not required to have any code that actually checks for collisions between objects; that's something we'll be adding later. For now, the program simply relies on the user to say when such events have occurred, using for example the '**k**' and '**c**' commands.

- You must follow standard Java coding conventions:
  - *class names always start with an upper case letter,*
  - *variable names always start with a lower case letter,*
  - *compound parts of compound names are capitalized (e.g., myExampleVariable),*
  - *Java interface names should start with the letter "I" (e.g., ISteerable).*

- Your program must be contained in a CN1 project called A1Prj. You must create your project following the instructions given at "2 – Introduction to Mobile App Development and CN1" lecture notes posted at Canvas (Steps for Eclipse: 1) File → New → Project → Codename

One Project. 2) Give a project name “A1Prj” and check “Java 8 project” 3) Hit “Next”. 4) Give a main class name “Starter”, package name “com.mycompany.a1”, and select a “native” theme, and “Hello World(Bare Bones)” template (for manual GUI building). 5) Hit “Finish”.). Further, you must verify that your program works properly from the command prompt before submitting it to Canvas: First make sure that the A1Prj.jar file is up-to-date. If not, under eclipse, right click on the dist directory and say “Refresh”. Then get into the A1Prj directory and type (all in one line): “java -cp dist\A1Prj.jar;JavaSE.jar com.codename1.impl.javase.Simulator com.mycompany.a1.Starter” for Windows machines (for the command line arguments of Mac OS/Linux machines please refer to the class notes). Alternatively, to prove it is working, you can also run the instructor provided program “RunAssignment.jar” (see class notes). Substantial penalties will be applied to submissions which do not work properly from the command prompt.

- You are not required to use any particular data structure to store the game world objects, but *all your game world objects must be stored in a single structure*. In addition, your program must be able to handle changeable numbers of objects at runtime – so you can’t use a fixed-size array, and you can’t use individual variables.

  Consider either the Java *ArrayList* or *Vector* class for implementing this storage. Note that Java Vectors (and ArrayLists) hold elements of type “Object”, but you will need to be able to treat the Objects differently depending on the type of object. You can use the “**`instanceof`**” operator to determine the type of a given Object, but be sure to use it in a polymorphically-safe way. For example, you can write a loop which runs through all the elements of a world Vector and processes each “movable” object with code like:

```
for (int i=0; i<theWorldVector.size(); i++) {
      if (theWorldVector.elementAt(i) instanceof IMovable) {
            IMovable mObj = (IMovable)theWorldVector.elementAt(i);
            mObj.move();
      }
}
```

- Points will be deducted for poorly or incompletely documented programs. Use of JavaDoc-style comments is highly encouraged, but not required.

- Students are encouraged to ask questions or solicit advice from the instructor outside of class. But have your UML diagram ready – it is the first thing the instructor will ask to see.

Deliverables

There are three deliveries for this assignment.

First Delivery: **Class Diagram. Due June 3rd, 2019**

Second Delivery – Object Creation: Test cases result + Source Files + Program Jar file. **Due June 5, 2019.**

Implementation: Commands 'a', 'y', 'b', 's', 'f', 'L', 'm', and 'p'

Third Delivery – Object Interaction: Test cases result + Source Files + Program Jar file. **Due June 10th, 2019**

Implementation: Commands '>', 'n', 'k', 'e', 'E', 'c', 'h', 'x', and 'l'

Except for the first delivery, the second and third delivery required *three steps* for submitting your program, as follows:

1. Be sure to verify that your program works from the command prompt.
2. Provide a result of running your test cases.
3. Create a *single* file in "ZIP" format containing (1) your UML diagram in .PDF format, (2) the entire "src" directory under your CN1 project directory (called A1Prj) which includes source code (".java") for all the classes in your program, and (3) the A1Prj.jar located under the "A1Prj/dist" directory which includes the compiled (".class") files for your program in a zipped format. Be sure to name your ZIP file as YourLastName-YourFirstName-a1.zip. Also include in your ZIP a text file called "readme" in which you describe any changes or additions you made to the assignment specifications.

    - Login to ***Canvas***, select "Assignment 1", and upload your *verified* ZIP file. Also, be sure to take note of the requirement stated in the course syllabus for keeping a *backup copy* of all submitted work (save a copy of your ZIP file).

Final notes:

(1) All submitted work must be strictly your own. It will be monitored closely!

(2) A grader will grade only one version of your work. Please check your work closely before submitting to Canvas.

Appendix – CN1 Notes

Input Commands

In CN1, since **`System.in`** is not supported, we will use a text field located on the form (i.e. the *Game* class) to enter keyboard commands. The **`play()`** method of *Game* will look like this (we will discuss the details of the GUI and event-handling concepts used in the below code later in the semester):

```
import com.codename1.ui.events.ActionListener;
import com.codename1.ui.Label;
import com.codename1.ui.TextField;
import com.codename1.ui.events.ActionEvent;
import java.lang.String;
private void play()
      {
      Label myLabel=new Label("Enter a
      Command:"); this.addComponent(myLabel);
      final TextField myTextField=new TextField();
      this.addComponent(myTextField);
      this.show();
      myTextField.addActionListener(new ActionListener(){

            public void actionPerformed(ActionEvent evt) {

            String sCommand=myTextField.getText().toString();
            myTextField.clear();
            switch (sCommand.charAt(0)){
                  case 'e':
                        gw.eliminate();
                        break;
                  //add code to handle rest of the commands
            } //switch
      } //actionPerformed
      } //new ActionListener()
      ); //addActionListener
      } //play
```

Random Number Generation

The class used to create random numbers in CN1 is **`java.util.Random`**. This class contains several methods including **`nextInt()`**, which returns a random integer from the entire range of integers, **`nextInt(int)`**, which returns a random number between zero (inclusive) and the specified integer parameter (exclusive), and **`nextFloat()`**, which returns float value (between 0.0 and 1.0). For instance, if you like to generate a random integer value between **`X`** and **`X+Y`**, you can call **`X + nextInt(Y)`**.

Output Strings

The routine **`System.out.println()`** can be used to display text. It accepts a parameter of type String, which can be concatenated from several strings using the "+"

operator. If you include a variable which is not a String, it will convert it to a String by invoking its *toString()* method. For example, the following statements print out "The value of I is 3":

```
int i = 3 ;
System.out.println ("The value of I is " + i);
```

Every CN1 class provides a *toString()* method inherited from **Object**. Sometimes the result is descriptive. However, if the *toString()* method inherited from **Object** isn't very descriptive, your own classes should override *toString()* and provide their own String descriptions – including the *toString()* output provided by the parent class if that class was also implemented by you.

For example, suppose there is a class **Book**, and a subclass of **Book** named **ColoredBook** with attribute *myColor* of type **int**. An appropriate *toString()* method in **ColoredBook** might return a description of a colored book as follows:

```
public String toString() {
      String parentDesc = super.toString();
      String myDesc = "color: " + "["  + ColorUtil.red(myColor) + ","
                                       + ColorUtil.green(myColor) + ","
                                       + ColorUtil.blue(myColor) + "]";
      return parentDesc + myDesc ;
}
```

(Abovementioned static methods of the **ColorUtil** class return the red, green, and blue components of a given integer value that represents a color.)

A program containing a **ColoredBook** called "**myBook**" could then display it as follows:

```
System.out.println ("myBook = " + myBook.toString());
```

or simply:

```
System.out.println ("myBook = " + myBook);
```

Number Formatting

The **System.out.format()** and **System.format()** methods supported in Java are not available in CN1. Hence, in order to display only one digit after the decimal point you can use **Math.round()** function:

```
double dVal = 100/3.0;

double rdVal = Math.round(dVal*10.0)/10.0;

System.out.println("original value: " + dVal);

System.out.println("rounded value: " + rdVal);
```

Above prints the following to the standard output

stream: original value: 33.333333333333336

rounded value: **33.3** (Please conform to 1 single decimal value).