CSUS COLLEGE OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science

CSc 133 – Object-Oriented Computer Graphics Programming
Fall 2022
Dr. Muyan

Assignment #1: Class Associations & Interfaces

Due Date: Monday, September 26th (11:59 PM)

Introduction

This semester we will study object-oriented graphics programming and design by developing a simple video game we'll call *WalkIt.* In this game, you (the player) will be controlling an ant walking around a path while trying to avoid collisions with spiders and keeping your ant fed.

The goal of this first assignment is to develop a good initial class hierarchy and control structure by designing the program in UML and then implementing it in Codename One (CN1). This version will use keyboard input commands to control and display the contents of a "game world" containing the set of objects in the game. In future assignments, many of the keyboard commands will be replaced by interactive GUI operations, and we will add graphics, animation, and sound. 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 console.

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 in the program encapsulates the notion of a ***Game***. A game in turn contains several components, including (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 manages 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 built-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 *WalkIt* game world size is fixed and covers 1000 (width) x 1000 (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 abstract kinds of game objects called: "fixed objects" of type ***Fixed*** with fixed locations (which are fixed in place) and "moveable objects" of type ***Movable*** with changeable locations (which can move or be moved about the world). For this first version of the game there are two concrete types that fixed objects are instantiated from which are called: ***Flag*** and ***FoodStation***; and there are two concrete types that moveable objects are instantiated from which are called: ***Ant*** and ***Spider***. Later we may add other kinds of game objects (both Fixed kinds and Moveable kinds) as well.

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

- All game objects have an integer attribute *size.* All game objects provide the ability for external code to obtain their size. However, they do not provide the ability to have their size changed once it is created. As will be specified in the later assignments, each type of game

object has a different shape which can be bounded by a square. The size attribute provides the length of this bounding square. All flags have the same size (chosen by you), assigned when they are created (e.g., size of all flags can be 10). You should also assign a size value to the ant. Sizes of the rest of the game objects are chosen randomly when created, and constrained to a reasonable positive integer value (e.g., between 10 to 50). For instance, size of one of the food stations may be 15 while size of another food station may be 20.

- All game objects have a *location*, defined by **`Point`** built-in class of CN1 (which resides under **`com.codename1.charts.models`** and uses **`float`** values), which is constituted from non-negative X and Y values that initially should be in the range 0.0 to 1000.0 and 0.0 to 1000.0, 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 flags and the ant, initial locations of all the game objects should be assigned randomly when created.

- 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). Initially, all objects of the same class have the same color (chosen by you), assigned when the object is created (e.g., flags could be blue, the ant could be red, food stations can be green, spiders can be black). 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.

- *Flags* are fixed game objects that have attribute *sequenceNumber*. Each flag is a numbered marker that acts as a "waypoint" on the path; following the path is accomplished by walking over on top of flags in sequential order (the flags mark the course of the path on the ground, see the diagram below). Flags are not allowed to change color once they are created. All flags should be assigned to locations chosen by you, when they are created.

- *FoodStations* are fixed game objects that have an attribute *capacity* (amount of food a food station contains). The initial capacity is proportional to the size of the food station. When the ant is hungry (i.e., running low on its food level), it must walk to a food station that is not empty before its food level becomes zero; otherwise it cannot move.

- All fixed game objects are not allowed to change location once they are created.

- Moveable game objects have integer attributes *heading* and *speed*. Telling a moveable object to *move()* causes the object to update its location based on its current heading and speed. The movable game objects move simultaneously according to their individual speed and heading. *Heading* is specified by a *compass angle* in degrees: 0 means heading north (upwards on the screen), 90 means heading east (rightward on the screen), etc. See below for details on updating an movable object's position when its *move()* method is invoked.

- Some movable game objects are *steerable*, meaning that they provide an interface called *ISteerable* that allows other objects to *change* their heading (direction of movement) after they have been created. Note that the difference between *steerable* and *moveable* is that

other objects can *request a change in the heading* of *steerable* objects whereas other objects can only request that a *movable* object update its own location according to its current speed and heading.

- *Ant* is moveable and steerable game object with attributes *maximumSpeed, foodLevel, foodConsumptionRate, healthLevel,* and *lastFlagReached*.

  The *maximumSpeed* of the ant is the upper limit of its *speed* attribute; attempts to accelerate the ant beyond its *maximumSpeed* are to be ignored (that is, the ant can never go faster than its *maximumSpeed*). You should set this value to a reasonable value.

  The *foodLevel* of the ant indicates how hungry it is; if the ant's food level is zero, it would have zero speed and cannot move. You should set this value to an initial reasonable value.

  The *foodConsumptionRate* of the ant indicates how much food the ant needs to consume each time the clock ticks. You should set this value to a reasonable value.

  The *healthLevel* of the ant starts at 10 and decreases each time the ant collides with a spider (see below for details). Heath level affects the performance of the ant as follows: if the ant has a perfect health (i.e., has health value 10), it can accelerate all the way up to its *maximumSpeed*; if the ant's health value is zero it would have zero speed and thus, cannot move at all; and if the ant's health value is between 10 and zero, it should be limited in speed to a corresponding percentage of their speed range (e.g., if the ant's health value is 5, it can only achieve 50% of its maximum speed). When the ant health is degraded because it is involved in a collision (see below for details), its speed is reduced (if necessary) so that this speed-limitation rule is enforced.

  The *lastFlagReached* indicates the sequence number of the last flag that the ant has reached in the increasing order.

  Initially, the ant should be positioned at the location of flag #1 (initially *lastFlagReached* is assigned to "1") and its heading should be assigned to zero and speed should be assigned to an appropriate positive non-zero value.

- Later we may add other kinds of game objects to the game which are steerable.

- Spiders are moveable (but not steerable) objects which walk on the ground the path is on. They add (or subtract) small random values (e.g., 5 degrees) to their heading while they move so as to not run in a straight line. If the spider's center hits a side of the world, it changes heading and does not move out of bounds. If the spider gets to the same location as the ant it decreases the health level of the ant by one. Spiders are not allowed to change color once they are created. Speed of a spider should be initialized to a reasonable random value (e.g., ranging between 5 and 10) at the time of instantiation. Heading of a spider should be initialized to a random value (ranging between 0 and 359) at the time of instantiation.

The preceding paragraphs imply several *associations* between classes: an inheritance hierarchy, interfaces such as for *steerable* objects, and aggregation associations between objects and where they are held. You are to develop a UML diagram for the relationships, and then implement it in CN1. Appropriate use of encapsulation, inheritance, aggregation, abstract classes, and interfaces are important grading criteria. Note that an additional important criterion is that *another programmer must not be able to misuse your classes*, e.g., if the object

is specified to have a fixed color, another programmer should not be able to change its color after it is instantiated.

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. Make sure to print your UML to a **single-page** pdf file in the **portrait** orientation (it is OK if the fonts and shapes look small on the pdf page since we can zoom-in to see the details). You are **not** allowed to use tools that automatically generate a UML from your code. Your UML must show all important associations between your entities (and built-in classes/interfaces that your entities extend/implement) and utilize correct graphical notations. For your entities you must use three-box notation (for built-in entities use one-box notation) 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 the game starts the player has three "lives" (chances to reach the last flag). The game has a *clock* which counts up starting from zero and the objective is to walk around the path to the last flag in the minimum amount of time.

The player uses keystroke commands to change the ant's heading (see below for details). The ant moves by its current speed in the direction it is currently heading each time the game clock "ticks" (see below).

The ant starts out at the first flag (#1). The player must walk the ant so that it intersects the flags in increasing numerical order. Each time the ant reaches the next higher-numbered flag the ant is deemed to have successfully walk that far along the path and its *lastFlagReached* field is updated. Intersecting flags out of order (that is, reaching a flag whose number is *more than* one greater than the most recently reached flag, or whose number is less than or equal to the most recently reached flag) has no effect on the game.

The food level of the ant continually goes down as the game continues. If the ant's food level reaches zero it can no longer move. The player must therefore occasionally walk the ant off the path to intersect with a food station, which has the effect of increasing the ant's food level by the capacity of the food station. After the ant intersects with the food station, the capacity of that food station is reduced to zero and a new food station with randomly-specified size and location is added into the game.

Collisions with spiders decrease the health level of the ant by one; if the health level of the ant reaches zero, it can no longer move.

If the ant can no longer move (i.e., its food or health level has reached zero), the game stops (the player "loses a life") and the game world is re-initialized (but the number of clock ticks is not set back to zero). When the player loses all three lives the game ends and the program exits by printing the following text message in console: "Game over, you failed!". If the ant reaches the last flag on the path, the game also ends with the following message displayed on the console: "Game over, you win! Total time: #", where # indicates the number of clock ticks it took the ant to reach the last flag on the path.

The program keeps track of following "game state" values: current clock time and lives remaining. Note that these values are part of the *model* and therefore belong in the *GameWorld* class.

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. The allowable input commands and their meanings are defined below (note that commands are case sensitive):

- 'a' – tell the game world to **a**ccelerate (increase the speed of) the ant by a small amount. Note that the effect of acceleration is to be limited based on *health level*, *food level*, and *maximum speed* as described above.

- 'b' – tell the game world to **b**rake (reduce the speed of) the ant by a small amount. Note that the minimum speed for the ant is zero.

- 'l' (the letter "ell") – tell the game world to change the *heading* of the ant by 5 degrees to the **l**eft (in the negative direction on the compass).

- 'r' – tell the game world to change the *heading* of the ant by 5 degrees to the **r**ight (in the positive direction on the compass).

- 'a number between 1-9'– **PRETEND** that the ant has collided with (walked over) the flag number **x** (which must have a value between 1-9); tell the game world that this collision has occurred. The effect of walking over a flag is to check to see whether the number ***x*** is exactly one greater than the flag indicated by *lastFlagReached* field of the ant and if so, update the *lastFlagReached* field of the ant by increasing it by one.

- 'f' – **PRETEND** that the ant has collided with a **f**ood station; tell the game world that this collision has occurred. The effect of colliding with a food station is to increase the ant's food level by the capacity of the food station (in this version of the assignment, pick a non-empty food station randomly), reduce the capacity of the food station to zero, fade the color of the food station (e.g., change it to light green), and add a new food station with randomly-specified size and location into the game.

- 'g' – **PRETEND** that a spider has **g**otten to the same location and collided with the ant. The effect of colliding with a spider is to decrease the health level of the ant as described above, fade the color of the ant (i.e., it becomes lighter red – throughout the game, the ant can have different shades of red), and (if necessary) reduce the speed of the ant so that above-mentioned speed-limitation rule is enforced. Since currently no change is introduced to the spider after the collision, it does not matter which spider is picked.

- 't' – tell the game world that the "game clock" has **t**icked. A clock tick in the game world has the following effects: (1) Spiders update their heading as indicated above. (2) all moveable objects are told to update their positions according to their current heading and speed, and (3) the ant's food level is reduced by the amount indicated by its *foodConsumptionRate*, (4) the elapsed time "game clock" is incremented by one (the

game clock for this assignment is simply a variable which increments by one with each tick). Note that all commands take immediate effect and not depend on 't' command (e.g., if 'a' is hit, the ant's speed value would be increased right away without waiting for the next 't' command to be entered).

- 'd' – generate a **d**isplay by outputting lines of text on the console describing the current game/ant state values. The display should include (1) the number of lives left, (2) the current clock value (elapsed time), (3) the highest flag number the ant has reached sequentially so far (i.e., *lastFlagReached*), (4) the ant's current food level (i.e., *foodLevel*), and (5) the ant's health level (i.e., *healthLevel*). All output should be appropriately labeled in easily readable format.

- 'm' – tell the game world to output a "**m**ap" showing the current world (see below).

- 'x'– e**x**it, by calling the method `System.exit(0)` to terminate the program. Your program should confirm the user's intent to quit before actually exiting. This means your program should force the user to enter 'y' or 'n' (see 'y' and 'n' commands below) to the text field (and hit enter) after (s)he enters 'x' to the text field (and hits enter). To accomplish this, consider setting a flag in your code whenever the last entered command from the text field is 'x'. Then, if the last entered command is 'x' and the user does not say 'y' or 'n' after saying 'x', you can print a message to the console asking for one of these commands and do not accept any other commands.

- 'y' – user has confirmed the exit by saying **y**es.

- 'n' – user has not confirmed the exit by saying **n**o.

Some of commands above indicate that you **PRETEND a collision happens**. Later this semester, your program will determine these collisions automatically[1].

The code to perform each command must be put in a separate method. When the *Game* gets a command which requires manipulating the *GameWorld*, the *Game* must invoke a method in the *GameWorld* to perform the manipulation (in other words, it is not appropriate for the *Game* class to be directly manipulating objects in the *GameWorld*; it must do so by calling an appropriate *GameWorld* method). The methods in *GameWorld,* might in turn call other methods that belong to other classes.

When the player enters any of the above commands, an appropriate message should be displayed in console (e.g., after 'b' is entered, print to console something like "break is applied").

Additional Details

- Method ***init()*** is responsible for creating the initial state of the world. This should include adding to the game world at least the following: a minimum of four *Flag* objects, positioned and sized as you choose and numbered sequentially defining the path (you may add more than four initial flags if you like - maximum number of flags you can add is nine); one *Ant*,

[1] In later assignments we will see how to actually detect on-screen collisions such as this; for now we are simply relying on the user to tell the program via a command when collisions have occurred. Inputting a collision command is deemed to be a statement that the collision occurred; it does not matter where objects involved in the collision actually happen to be for this version of the game as long as they exist in the game.

initially positioned at the flag #1 with initial heading which is zero, initial positive non-zero speed as you choose, and initial size as you choose; at least two *Spider* objects, randomly positioned with a randomly-generated size, heading, and speed; and at least two *FoodStation* objects with random locations and with random sizes.

- All object initial attributes, including those whose values are not otherwise explicitly specified above, should be assigned such that all aspects of the gameplay can be easily tested (e.g., spiders should not move so fast that it is impossible for them to ever cause a harm to the ant).

- In this first version of the game, it is possible that some of the abstract classes might not include any abstract methods (or some classes might not have any fields/methods). In future assignments, we may add such class members as needed.

- In order to let a child class set a private field defined in its parent that does not provide a proper public setter for the field, consider having a constructor in the parent that takes in the value to be set as a parameter (e.g., **`public MyParentClass(int i) {myField = i;…}`**) and calling this constructor from the child's constructor (e.g., **`public MyChildClass(int value) {super(value);…}`**, note that the value is passed as a parameter to child constructor instead of being generated there). If the field resides in the parent of the parent class, consider having the proper constructor in that class (e.g. **`public MyGrandParentClass(int i) {myField = i;…}`**) and calling it from the constructor of the parent class (e.g., **`public MyParentClass(int i) {super(i);…}`**) to pass the value coming from the child to the grandparent.

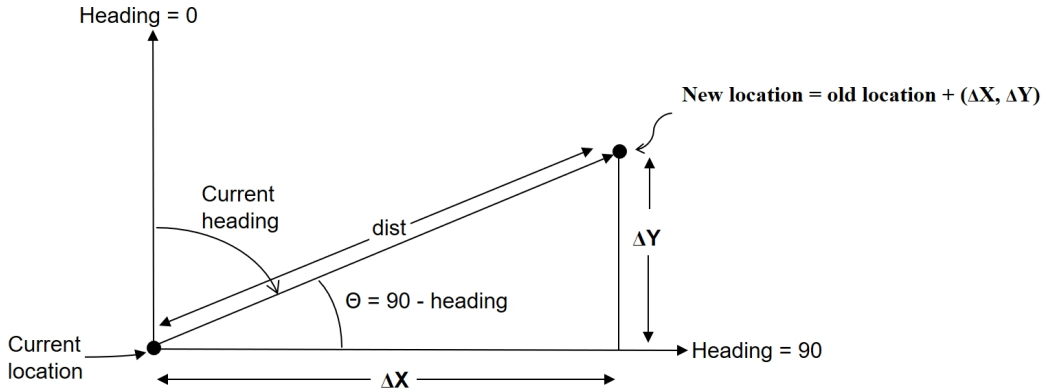
- If a setter is defined in a parent class and you want to prevent the child to have an ability to set that value, consider overriding the setter in the child with the method that has empty body implementation (e.g., **`public void setX(int i){}`**)

- All classes must be designed and implemented following the guidelines discussed in class, including:
  - *All data fields must be private.*
  - *Accessors / mutators must be provided, but only where the design requires them.*

- Moving objects need to determine their new location when their *move()* method is invoked, at each time tick. 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 where **`θ`** = 90 - heading (90 minus the heading). Below diagram shows the derivation of these calculations for an arbitrary amount of time; in this assignment we are assuming "time" is fixed at one unit per "tick", so "elapsed time" is 1 second (1000 milliseconds):



$$dist = rate \times time \quad = \quad \text{speedInUnitsPerSecond} \times \frac{\text{elapsedMilliSecs}}{1000}$$

$$\cos\theta = \frac{\Delta X}{dist};\ \text{so}\quad \Delta X = \cos\theta \times dist.\quad \text{Likewise, } \Delta Y = \sin\theta \times dist$$

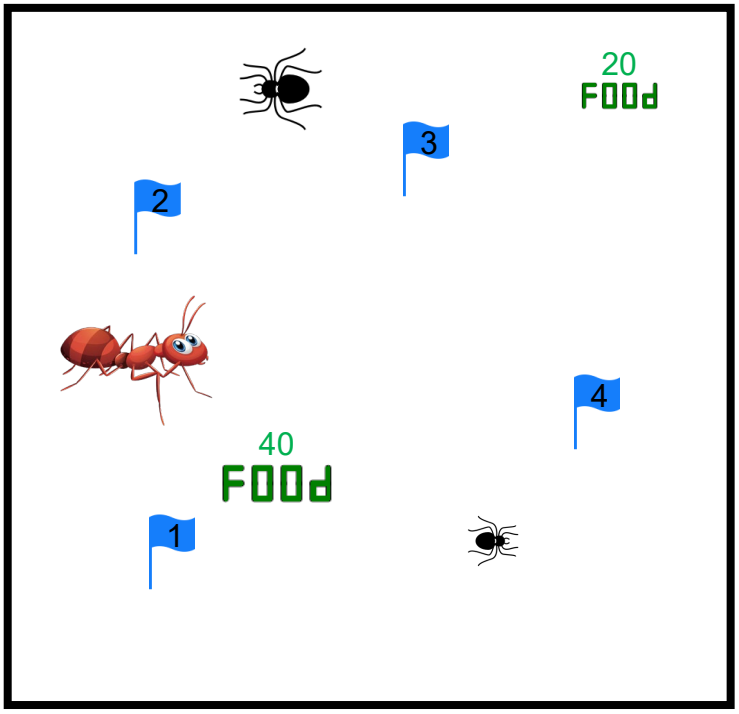
You can use methods of the built-in CN1 ***Math*** class (e.g., **`Math.cos()`**, **`Math.sin()`**) to implement the above-listed calculations in *move()* method. Be aware that these methods expect the angles to be provided in "radians" not "degrees". You can use **`Math.toRadians()`** to convert degrees to radians.

- For this assignment <u>all</u> output will be in text form 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, as the following:

```
Flag: loc=200.0,200.0 color=[0,0,255] size=10 seqNum=1
Flag: loc=200.0,800.0 color=[0,0,255] size=10 seqNum=2
Flag: loc=700.0,800.0 color=[0,0,255] size=10 seqNum=3
Flag: loc=900.0,400.0 color=[0,0,255] size=10 seqNum=4
Ant: loc=180.2,450.3 color=[255,0,0] heading=355 speed=50 size=40
     maxSpeed=50 foodConsumptionRate=2
Spider: loc=70.3,70.7 color=[0,0,0] heading=45 speed=5 size=25
Spider: loc=950.6,950.3 color=[0,0,0] heading=225 speed=10 size=20
FoodStation: loc=350.8,350.6 color=[0,255,0] size=15 capacity=15
FoodStation: loc=900.0,700.4 color=[0,255,0] size=20 capacity=20
```

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 "Appendix – CN1 Notes" below). Please see "Appendix – CN1 Notes" below for also tips on how to print one digit after a decimal point in CN1.

***<u>For this assignment, the only required depiction of the world is the text output map as shown above</u>*. Later we will learn how to draw a *graphical* depiction of the world, which then for the above map might look something like the image shown below** (the pictures in the image are not all precisely to scale, and note that this is only to give you an idea of what the above map represents – your program is *not* required to produce any graphical output like this in the later assignments).



- You are not required to use any particular data structure to store the game world objects. However, your program must be able to handle changeable numbers of objects at runtime; this means 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 your storage will hold elements of type GameObject, but you will need to be able to treat them 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 theWorldVector and processes each "movable" object with code like:

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

- You can utilize **`java.util.Random`** class (see the "Appendix – CN1 Notes" below) to generate random values specified in the requirements (e.g., to generate initial random sizes and locations of objects or to pick an object randomly for "pretend" commands).

- It is a requirement for *all* programs in this class that the source code contain *documentation*, in the form of comments explaining what the program is doing, including comments describing the purpose and organization of each class and comments outlining each of the main steps in the code. Points will be deducted for poorly or incompletely documented programs. Use of JavaDoc-style comments is highly encouraged.

- It is a requirement to 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 uncheck "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. Then get into the A1Prj directory and type (all in one line) the following for Windows machines:

  ```
  java -cp dist\A1Prj.jar;JavaSE.jar com.codename1.impl.javase.Simulator com.mycompany.a1.Starter
  ```

  For the command line arguments of Mac OS/Linux machines please refer to the class notes. Penalties will be applied to submissions which do not work properly from the command prompt.

Deliverables

There are *three steps* which are required for submitting your program, as follows:

1. Create a "TEXT" (i.e., not a pdf, doc etc.) file called "readme-a1.txt" that includes the **remote** lab number and the name of the specific machine you have used in that lab to build/test your program. Be sure to **verify that your program works from the command prompt on the remote lab machine** as explained above. For instructions on how to connect to a remote lab machine, please see the related page on Canvas and for hints on how to build/test your assignment on a lab machine, please see the syllabus. In addition, **be sure that you include the *src* folder and *jar* file generated/tested on the remote lab machine in the below-mentioned zip file**. You may also include additional information you want to share with the grader in this text file. You will receive the grader comments on your text file when grades are posted (click on "View Feedback" link next to readme-a1.txt on Canvas to see the comments).

2. Create a "ZIP" file 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 is automatically generated by CN1 and includes the compiled (".class") files for your program in a zipped format. Do **NOT** include other directories/files under the CN1 project directory. Be sure to name your ZIP file as YourLastName-YourFirstName-a1.zip.

3. Login to ***Canvas***, select "Assignment#1", and upload your TEXT file and ZIP file separately (**do NOT place this TEXT file inside the 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 TEXT and ZIP files).

All submitted work must be strictly your own!

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();
            if(sCommand.length() != 0)
                  switch (sCommand.charAt(0)) {
                        case 'a':
                              gw.accelerate();
                              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 **String.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
```