CSUS COLLEGE OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science

CSc 133 – Object-Oriented Computer Graphics Programming
Fall 2012 – John Clevenger

Assignment #1: Class Associations & Interfaces

Due Date: Thursday, September 27th [2+ weeks]

Introduction

This semester we will be studying object-oriented computer graphics programming by developing a program which is a variation of a classic arcade and home computer game called “Frogger”. In this game you will be controlling a frog which is trying to hop its way through a series of hazards to safety. You are to write a program which uses keyboard input to control a frog hopping through a world divided into horizontal sections of various kinds: grass, traffic lanes containing moving vehicles, a dirt river bank containing rocks, river channels containing floating objects, and lily pads on the far side of the river. The frog starts at the bottom (on the grass area), and has to move across the traffic lanes, river bank, and river channels to arrive at the safety of one of the lily pads. The frog can land on asphalt (in the traffic lanes), on dirt on the river bank, on logs or turtles floating it the river, or on lily pads at the far side. The frog *cannot* land on a place where there is traffic on the road (too dangerous), or on rocks on the river bank (too hot), or in the water (he can’t swim….).

If you have never seen Frogger, you can play the original arcade version at http://www.classicgamesarcade.com/game/21607/frogger.html. A Flash version is available at http://www.happyhopper.org/; there are also many other versions available on the web (Google “frogger”), and you can even download Frogger for your iPhone or Android from the appropriate app store. (It is not necessary to be familiar with the original game to do any of the assignments during the semester – but seeing how the original game plays might help you understand the general idea of what we’re going to be doing.)

The initial version of your game will be *text-based*. As the semester 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 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, 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 game state variables, and (2) a set of methods to accept and execute user commands. Later, we will learn that a component such as *GameWorld* which 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.

The program also has a class named ***Starter*** which has the **`main(String[]args)`** method. **`main()`** does one thing – construct an instance of the ***Game*** class. The game constructor then instantiates the game components, and starts the game by calling a method named **`play()`**. The **`play()`**method then accepts keyboard commands from the player and invokes appropriate methods to manipulate the game world and display the game state.

The following shows the pseudo-code implied by the above description.

```
class Starter {
   main() {
      Game g = new Game();
   }
}
```

```
class GameWorld {
   // code here to hold and
   // manipulate world objects
   // and related game state data
}
```

```
class Game {
   private GameWorld gw;

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

   private void play() {
      // code here to accept and
      // execute user commands that
      // operate on the GameWorld
   }
}
```

Game World Objects

For this version of the game, the **`GameWorld`** is an area 1000 units wide by 500 units high. The game world is divided into ten horizontal rows, each of which is 50 units high and represents a different section of the world which the frog must traverse: the bottom row is grass; each of the next 4 rows is a traffic lane (asphalt); the next row is a river bank (dirt); each of the next three rows is a flowing river channel (water), and the top row contains lily pads in some locations and water in the rest. Traffic in each traffic lane moves in the opposite direction from that of its neighboring lane(s), and at different speeds. Each of the three river channels flows from west (left) to east, but at different speeds.

Most of the game world is empty; however, some locations in the world are occupied by "game objects". The game world maintains a collection of the game objects currently in the world. There are two abstract kinds of game objects: "fixed game objects" (which always stay at the same place in the world), and "moving game objects" (which move around in the world automatically). There are two concrete kinds of fixed objects: *rocks* and *lily pads*. Some locations in the river bank row have rocks in them (while the remaining locations are empty dirt), and some locations in the top row have lily pads in them (while the remaining locations are water). There are two abstract kinds of moving objects: *floaters* and *vehicles*. There are two concrete kinds of floaters: *logs* and *turtles*; and there are two concrete kinds of vehicles: *cars* and *trucks*. In addition, there is one other concrete kind of game object: frogs.

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

- All game objects have a *location*, defined by floating point values X and Y. The point (X,Y) is the center of the object. The origin of the "world" is the lower left hand corner of the screen (although we will change this later; think of it this way for now). All game objects must provide the ability for other objects to obtain their location. Some game objects have changeable locations (that is, they are moveable), while others have locations that are not allowed to change. The program must properly implement this requirement.

- All game objects have a *color*, defined by a value of Java type **`java.awt.Color`**. All concrete objects of the same class start with the same color (e.g., all frogs are green). All game objects provide the ability for other objects to obtain their color. Some types of game objects have color which is changeable (meaning that they provide an interface which allows other objects to modify their color), while other objects have a color which cannot be changed once the object is created. Objects which are not explicitly stated as having the ability to change their color must not support that ability.

- Some game objects are *moveable*, meaning that they have attributes *speed* and *direction* (one of North, South, East, or West) and that they provide an interface that allows other objects to control their movement: telling a moveable object to *move()* causes the object to update its location based on its current speed and direction.

- Some game objects are *hoppable*, meaning that they provide an interface that allows other objects to instruct them to *hop in a specified direction*. Telling a hoppable object to hop in a certain direction causes the object to change its location by a fixed amount in the specified direction; the amount which it hops is exactly the same as the width of a lane. Note that the difference between *hoppable* and *moveable* is that other objects can *specify the direction of hopping* for *hoppable* objects whereas other objects can only request that a *movable* object update its location according to its current internal speed and heading.

- Floating objects (floaters) have an attribute *buoyancy* which controls how much weight they can support. Objects with zero buoyancy stop floating (sink to the bottom). When a floating object is created it is randomly assigned to one of the three river channels (meaning its Y location is determined and remains fixed thereafter). Floating objects always move in the direction and at the speed of the river channel in which they are floating (and they never change channels).

- Logs are floating objects which have a *length* attribute, which will be either "short" or "long". Logs start with a random positive buoyancy which never changes.

- Turtles are floating objects which have a *size* attribute, which will be either "large" or "small". Turtles have a buoyancy which decreases over time (because the turtles eventually dive below the surface). Turtles can *change color*, which they do when they are about to dive.

- Vehicles have a unique Vehicle Identification Number (VIN); this is a positive integer which is different for every vehicle in the game. For simplicity the first vehicle gets VIN 1, then each new vehicle after that gets the next available integer. Car vehicles also have an integer attribute *passengerCount* specifying how many passengers the car can carry, while Trucks have an attribute *length* specifying the length of the truck.

- Rocks are fixed objects which have an attribute *size*; for this assignment there are just two sizes (small and large).

- Lily pads are fixed objects which have an attribute *height*; for this assignment there are just two heights (tall and short). Lily pads start out empty; as the game progresses frogs try to hop on them (but only one frog is allowed per lily pad).

- Frogs are game objects which are both *moveable* and *hoppable*. Telling a frog to *move()* updates its location as described above under movable objects. Telling a frog to *hop* causes it to change its location as defined by the *hoppable* interface and also has the side effect of changing its *speed* to zero.

The preceding paragraphs imply a certain set of *associations* between classes – an inheritance class hierarchy among game objects, interfaces presented by certain classes (e.g. for *changeable color, moveable,* and *hoppable* objects), and aggregation associations between objects and where they are held. You must develop a UML diagram for the relationships, and then implement it in a Java program. Appropriate use of encapsulation, inheritance, aggregation, abstract classes, and interfaces are important grading criteria.

Game Play

The playing field is a rectangular area divided into ten horizontal lanes as previously described. The top row starts with five lily pads, equally spaced along the row. The player gets five frogs, one at a time, and tries to "hop" them safely from the bottom row across the hazards to the safety of an empty lily pad. Frogs can be "hopped" in one of four directions (NORTH, SOUTH, EAST, or WEST) as long as doing so does not cause it to leave the playing area (the frog dies if this happens). Hopping safely onto each successive lane earns some points (you may decide how many). Hopping onto a hazardous area (a traffic lane position occupied by a vehicle, dirt occupied by a rock, a river channel with no log or turtle to support it, or a lily pad already occupied by a previous frog) or riding on a turtle which dives, causes the frog to die.

The game world also keeps track of three "game state" values: elapsed time, frogs remaining, and current score. The objective is to obtain the highest score in the least amount of elapsed time.

Commands

Once the game world has been created and initialized, the game constructor should call method **`play()`** to actually begin the game. **`play()`** repeatedly calls a method named **`getCommand()`**, which prompts the user for single-character *commands*. Commands should be input using the Java InputStreamReader, BufferedReader, or Java Scanner class (see the "Java Coding Notes" Appendix at the end, and also page 23 in the text).

The allowable input commands and their meanings are defined below. Any undefined or illegal input should generate an appropriate error message on the console and ignore the input. Each command returned by **`getCommand()`** should invoke a corresponding function in the game world, as follows:

'**r**' – add a new **r**ock to the world with a size chosen randomly (small or large) and a location chosen randomly along the dirt river bank lane.

'**l**' (the letter 'ell') – add a new **l**og to the world with a randomly-selected length, and a randomly-selected X location within one of the three river channels (selected at random). The speed of a log always matches that of the chosen river channel.

‘**u**’ – add a t**u**rtle to the world with random size and buoyancy attributes.

‘**c**’ – add a new **c**ar to the world, with a randomly chosen traffic lane and a passenger capacity in the range 1-8. All vehicles in a given traffic lane have the same speed and heading.

‘**k**’ – add a new truc**k** to the world, with a randomly chosen traffic lane and length.

‘**f**’ – add a new **f**rog to the world at a randomly-chosen X location along the bottom (grass) lane.

‘**n**’, ‘**s**’, ‘**e**’, ‘**w**’ – hop the frog **n**orth, **s**outh, **e**ast, or **w**est (respectively) one lane width.

‘**1**’ – pretend that the frog has collided with a Car (and therefore dies and is removed).

‘**2**’ – pretend that the frog has collided with a Truck (and therefore dies and is removed).

‘**3**’ – pretend that the frog has hopped onto a hot rock (and therefore dies and is removed).

‘**4**’ – pretend that the frog has hopped onto a log. The effect of this is that the frog’s speed and direction are set to match those of the log. Later we will see how to associate a specific log’s attributes with the frog; for this assignment you should just choose any log at random and assign that log’s speed and direction to the frog.

‘**5**’ – pretend that the frog has hopped onto a turtle. The effect of this is that the frog’s speed and direction are set to match those of the turtle. As with logs, for now you should just pick a turtle at random and assign it’s attributes to the frog.

‘**6**’ – pretend that the frog has hopped onto an unoccupied lily pad. The effect of this is that the frog’s position matches that of the lily pad, it’s speed is set to zero, and it remains there for the rest of the game; and the player earns some additional points (you may decide how many). For now you should choose any unoccupied lily pad for this; later we’ll see how to associate a specific lily pad with a frog.

‘**7**’ – pretend that the frog has hopped onto an occupied lily pad. The effect of this is that the frog dies (from fighting with the previous occupant) and is removed.

‘**8**’ – pretend that the frog has hopped into the water. The effect of this is that the frog dies (he drowns; he can’t swim, remember?) and is removed. Note that this could occur in the game due to the frog hopping directly into the river, or due to its riding on a turtle which dives. Your program for this assignment does not need to distinguish between these states; we’ll see how to do that in a future assignment.

‘**9**’ – pretend that the frog has attempted to move outside the game world boundaries (i.e. past the North/South/East/West borders). The effect of this is that the frog dies and is removed. Note that in the game this could happen for a variety of reasons: hopping out of bounds, or riding a log or a turtle out of bounds for example. Your program for this assignment does not need to distinguish between these states.

‘**t**’ – tell the game world that the “game clock” has ticked. A clock tick in the game world has the following effects: (1) all moveable objects are told to update their positions according to their current heading and speed; (2) every turtle’s buoyancy is reduced by one; (3) any turtles with a new buoyancy less than four change color; (4) any turtles with a new buoyancy of zero “dive” and are removed from the game; and (5) the “elapsed game time” is incremented by one.

‘**d**’ – print a **d**isplay (output lines of text) giving the current game state values, including: (1) the current game clock value, (2) the current score, and (3) the number of frogs left to be played. 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 actually 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:
    - o *All data fields must be private,*
    - o *Accessors / mutators must be provided, but only where the design requires them,*
    - o *Appropriate use of abstract classes must be made,*
    - o *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.*
- 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, one line for each object in the world. Thus the output for a single ‘m’ command might appear similar to:

```
LilyPad: x=130,y=475 color=[255,255,0] height=short...
LilyPad: x=900,y=475 color=[255,255,0] height=tall...
<three more top-row LilyPads here...>
Rock: x=400,y=275 color=[128,128,128] size=small...
Log: x=100,y=325 color=[0,0,0] speed=10 dir=EAST buoy=6...
Log: x=840,y=425 color=[0,0,0] speed=5 dir=EAST buoy=4...
Turtle: x=320,y=375 color=[255,0,255] speed=15 buoy=1...
Turtle: x=3600,y=375 color=[128,0,0] speed=15 buoy=4...
Frog: x=130,y=475 color=[0,255,0]
Frog: x=600,y=25 color=[0,255,0]
```

  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 for additional details.

- Note that it is *not* appropriate for the program to maintain some kind of array (or other data structure) representing the entire world area. That is, the program should not declare an array of size 1000x500 just because that is the size of the world. The world is represented by the collection of objects in it (each of which has a location), not by a structure containing some kind of element for each location in the world independent of what is (or is not) located there.

- Each vehicle is required to have a unique Vehicle Identification Number (VIN). Since vehicles are constructed by invoking the Car and Truck constructors, there has to be some way to coordinate VIN usage such that no two vehicles – a car and a truck, for example, acquire the same VIN. This is an appropriate place to use the Java notion of a *class-wide (static) variable.* If you create a static variable *nextAvailableVIN* in the Vehicle class and have the constructors for Car and Truck both use that static variable (from their parent class) to obtain their local VIN value, every VIN will be unique – even between cars and trucks. Note: you must also arrange that, once a vehicle has been created, it is not possible to *change* its VIN.

- The program must handle any situation where the player enters an illegal command – for example, a command to hop onto a log when there are no logs in the world – by printing an appropriate condition-specific error message on the console (for example, "Cannot execute '4' – no logs exist in the world") 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 the numeric commands.

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

- Your program must be contained in a Java *package* named "**`a1`**" ("a-one", lower-case). Specifically, every class in your program must be defined in a separate **`.java`** file which has the statement "**`package a1;`**" as its first statement. Further, it must be possible to execute the program from a command prompt by changing to the directory containing the **`a1`** package and typing the command: "**`java a1.Starter`**". <u>Verify for yourself that this works correctly</u> from a command prompt before submitting your program. Note that the appropriate use of *sub-packages* within package "**`a1`**" is both acceptable and encouraged. For example, it might be logical to group all the classes defining the GameObject hierarchy in a subpackage named **`gameObject`**. See pages 16-18 and 29-31 in the text for more on the use of Java packages.

- Single keystrokes don't invoke action -- the human hits "enter" after each key command.

- 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 <u>single</u> 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. For example, classes and interfaces should have header comments explaining their purpose/function; methods should have header comments explaining their operation and return values, and should have internal comments explaining what each block of statements does. 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

Submission is to be done using the ***SacCT***. Submit a single “ZIP” file containing (1) your UML diagram in PDF format, and (2) the *Java* source code for all the classes in your program and also (3) the *compiled (“.class”)* files for your program. Be sure your submitted ZIP file contains the proper subpackage hierarchy. Also, be sure to take note of the requirement (stated in the course syllabus) for keeping a *backup copy* of all submitted work. If you are not familiar with submitting assignments using SacCT, see the paper in the Handouts section of my 133 web page.

All submitted work must be strictly your own!

Appendix – Java Coding Notes

Input Commands

In Java 5.0 and higher, the **`Scanner`** class will get a line of text from the keyboard:

```
     Scanner in = new Scanner (System.in);
     System.out.print (“Input some text:”);
     String line = in.nextLine();
  or int aValue = in.nextInt();
```

Random Number Generation

The class used to create random numbers in Java is **`java.util.Random`**. This class contains methods **`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 **`nextBoolean()`**, which returns a random boolean value (either true or false). The Random class is discussed on pg. 28 of the text.

Output Strings

The Java 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 Java class provides a *toString()* method. Sometimes the result is descriptive; for example, an object of type **`java.awt.Color`** returns a description of the color. However, if the *toString()* method is the default one inherited from Object, it 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 **`Ball`** with attribute *radius*, and a subclass of **`Ball`** named **`ColoredBall`** with attribute *myColor* of type java.awt.Color. An appropriate *toSring()* method in **`ColoredBall`** might return a description of a colored ball as follows:

```
public String toString() {
     String parentDesc = super.toString();
     String myDesc = "ColoredBall: " + myColor.toString();
     return parentDesc + myDesc ;
}
```

A program containing a **`ColoredBall`** called “**`myBall`**” could then display it as follows:

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

or simply:

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

JC:jc
9/10/12