Extreme Frogger Program Specification
Jake Shapiro
Matai Curzon
CS 112
Scott Alfeld
3 May 2019

Our program is an improvement of the classic game Frogger. It uses the base mechanics of Frogger, where the goal is to simply cross from the bottom of the screen to the top, but makes it more fun and interesting by adding enemies which you must evade or shoot, biomes and temperature mechanics, and by making the game endless by procedurally generating new levels and challenges.

When a user begins our program, the game will launch and the user will use the keyboard to move their character to avoid obstacles, and click the mouse to shoot enemies, and interact with the world. The program will respond to input by moving the player's character (and determining how it will then interact with other objects based on its new location) and by responding to the clicks to shoot projectiles at enemies who chase the player. It is a game.

Classes

Frogger: This is the main class of the game, and it contains the main method. Every time the game is launched, one instance of Frogger is created, and the entire game is run within this instance. Class Frogger is based off of KeyboardSpheres from lab 08. It used code from the lab to generate a world and start a runner thread. The Runner thread contains a call to update the world continuously as long as the program is running. Frogger extends JPanel.

Variables:

1. final int WIDTH and HEIGHT: The dimensions of the window in which the game is played. These are public static final because they are the same for every frogger, accessed universally, and immutable
2. int FPS: This int is public, static, and final and is used in making the animation run smoothly
3. World world: world is a variable of type World. It is the current "level." Every Frogger always contains one World. More details in world class
4. int livesRemaining: this int is the number of chance the player has before he dies and a new world is created. It is public because we saw no need to privatize it

5. int score: This int is the player's current score. It is added to whenever the players gets to the top of the screen. It is public, but, like livesRemaining, could easily be made private if we saw a need.
6. int clickX and clickY: int coordinates of the most recent click. This was the easiest place to store them that provided maximum accessibility from other locations in the code.
7. Thread mainThread: This is the mainThread where the game is run. We created threading to be able to animate and update simultaneously like in the labs we did.
8. Boolean gameOver and started: these booleans are self-explanatory. The game behaves differently if it has not been started or the player is dead

Constructor:

The constructor for Frogger calls methods to read images of game components into the game, creates the initial world, activates the key and mouse listener, creates the Runner that repeatedly updates the game and starts the main thread. This is only ever called one time.

Methods:

I. run(): This is the main loop of the game and calls update methods, methods to check for collisions, methods that run the temperature timer, repaint(), and a call to thread.sleep to slow down the graphics and the gameplay
II. kill(): This void method resets the world, subtracts a life from the player, and momentarily pauses the game. It is invoked any time the frog is killed
III. checkCollision(): This boolean method determines the Segment (defined later in its own class) the frog is in and checks to see if it occupies the same location as one of the obstacles also contains in that Segment
IV. checkLivesEaten(): This boolean method returns true if the frog occupies the same location as one of the extra lives on the screen
V. Mouse and Keyboard Listener Methods: These void methods are included to read clicks and movements from the mouse and keyboard
VI. start(): This method sets the game in motion by resetting the world and changing the started and gameOver booleans
VII. Main(String[] args): Main simply creates the graphics window and creates a Frogger
VIII. paintComponent(): This calls the draw methods of all of the components ever shown on screen. It also contains calls to drawInstructions(), drawStartScreen(), and drawEndScreen(), all of which are void and do exactly what their names imply

World: *Behind Frogger, World is the second most important class in the game. A World is the current level the player is on. It contains the 15 segments that make up the map, the enemies, the projectiles, and the frog. Every world has a biome which changes how it is drawn and may*

contain temperature mechanics. A new world is created whenever the frog dies or completes the current world.

Variables:

1. Frog frog: Every world contains one frog, which is the representation of the player within the world
2. ObjectHolder segments: This is an ObjectHolder of the segments that comprise the world
3. String biome: this is the biome of the world
4. ArrayList enemies and projectiles: ArrayLists that contain the active enemies and projectiles in the world
5. ObjectHolder lives: This ObjectHolder contains the extra lives that the frog can acquire. Currently, we do not have these implemented because they made the game too easy, but adding them in only requires uncommenting the lines of code in the constructor that adds them to each world
6. Static boolean enemiesAllowed: this variable determines whether or not enemies are to be added to the world (note: the enemies ArrayList is always constructed, just not necessarily filled). It is static so you can play an entire game of Frogger and have every new world have the same setting.

Constructor:
The constructor for world creates a new ObjectHolder (our generic object holder) of segments (Roads and SafeAreas) while assigning them y coordinates based on simple math, creates a Frog, randomly selects a biome, creates the ArrayList that stores projectiles, and creates the ArrayList that stores enemies and adds three enemies into the ArrayList.

Key Methods:

I. updateEnemies(): This boolean method calls the update method of each enemy (which is in their own class) and then checks to see if they now occupy the same space as the frog and are still alive, returning true if they are and false otherwise.

II. checkEnemiesShot(): This void method has a nested for loops which checks the location of every projectile against that of every enemy. If a projectile is within the hurtbox of an enemy, that enemy's boolean shot variable is set to true. Enemies are invisible if they are shot, but not actually removed from the game. One new enemy is made to replace each one that is shot. This method threw a null pointer exception one time, the reason for which we could not find, so now most of it is contained in a try-catch statement to prevent it from crashing the program again.

III. Class world also contains methods that are called from the drawing method in frogger such as drawSegments(Graphics g, String biome), that simply call other methods that do the actual drawing.

Enemy: *Enemies exist within the world and are represented visually by the birds chasing the frog. They have positions and velocities, and their function is to chase and try to catch (occupy the same location) as the frog. There are always three live enemies in the world, and there can be more dead ones. Enemies, when killed, are not removed from the, rather their shot variable is set to true, and they are not drawn and incapable of killing the frog. Because all existing enemies are removed from the game when a new world is created, dead enemies are purged often enough that we never need to worry about having too many in the game.*

Variables:

1. int size: This int stands for the physical size of the enemy's hitbox that the Frog can't touch without dying.
2. boolean shot: A boolean for checking whether an enemy has been shot or not. Shot enemies cannot harm the frog and are not drawn.
3. static double velocity: This double holds the velocity that the enemy travels at towards the Frog.
4. static BufferedImage enemyDown: A BufferedImage that holds the .png file called "enemyDown.png" for when the enemy is going down screen.
5. static BufferedImage enemyUp: A BufferedImage that holds the .png file called "enemyUp.png" for when the enemy is going up screen.

Constructor: There are two constructors for Enemy. One constructor is used to spawn 3 enemies each time a new World instance is created. This constructor takes no parameters and creates three new enemies between 0 and 200 pixels from the top and all across the width. It sets the position and sets health to 10. The other constructor takes in two int parameters and is used when respawning an enemy after it has been killed.

Methods:

1. draw(Graphics G): Draws the enemy object with either the enemyUp.png or enemyDown.png depending on the velocity.
2. readImage(): reads the necessary .png files and stores them in a BufferedImage variable.
3. update(int frogX, int frogY): Finds the angle (theta) from the position of the Enemy object to the Frog object. Then it multiplies cos(theta) by the x velocity to get the correct x velocity and sin(theta) by the y velocity to get the correct y velocity.

Frog: *Frog objects are the representation of the player on the screen. There is only one frog active at a time, and a new one is created for each world. The class also contains the code that handles the temperature mechanics of the tundra and desert biomes.*

Variables:

1. int xCoordinate: Holds the x coordinate of the frog.
2. int yCoordinate: Holds the y coordinate of the frog.
3. final int sizeX=50: Size of the frog hitbox.
4. final int sizeY=50: Size of the frog hitbox.
5. BufferedImage: We have multiple BufferedImage variables to hold 5 variations of the frog, jumping right, jumping left, jumping up, jumping down, and standing still. We also have BufferedImages to hold .png files for ice and fire that are used in conjunction with biomes and the movement timer.
6. int count: Count determines how many frames each jump image stays rendered on screen. It's basically used as a timer.
7. Booleans left, right, up, down: A boolean holding which way the frog jumped in order to use the correct jumping image.
8. int movementTimer: The actual value of the movementTimer at any time during the instance of Frogger.
9. final int defaultTimerValue = 400: Sets the initial timer value to 400.
10. int timerResetValue=150: Sets the minimum timer value to 150.
11. booleans frozen, burned: Booleans determining whether the frog has died from freezing or burning.

Constructor:

The constructor for Frog sets its location to the bottom of the screen, reads in its images, and sets its movement timer to the default initial value.

Methods:

I. jump(char c): Takes in a character (w,a,s,d) and advances the frog in the respective direction. If the frog jumps off screen, the method pulls it back on screen. The method also sets booleans used to draw the correct jumping image and sets the count variable to 10.

II. draw(Graphics g, String Biome): The method continuously draws .png for the frog object. It takes in a biome to decide whether or not it needs to call the drawMovement() method. Whenever the count is >10, it will draw a variation of the jumping frog image (depends on which way the frog jumps). Whenever count <10, it draws the standard, still image of the sitting frog.

III. **Temperature Management Methods**

A. getTime(): returns the time on the timer, used when drawing the timer

B. resetTimer(): resets the timer to timerResetValue, and lowers the reset value by 4 if it is greater than or equal to 154. This method means that after a certain number of jumps, the frog will only have 150 cycles before catching on fire, creating more urgency as his little frog feet get hotter and more burnt
C. tickTimer(): Decrements the timer by one every cycle of Frogger
D. checkTimer(String biome): checks to see if the timer has run out and setting the frog to be burned or frozen, depending on biome
E. drawMovementTimer(): draws a bar on screen that gets smaller as the timer runs out
F. drawTemp(Graphics g, String biome): Provides visual indication that the timer is almost out by drawing either ice or fire behind the frog and giving a textual warning that the frog is about to succumb to the elements.

Life: *Life implements drawable, so it can be stored in an ObjectHolder. Our game supports adding in extra lives. We removed them right before finalizing the game because it made it too easy, but re-enabling them is as simple as un-commenting two lines of code in the constructor for World. Lives are stored in the world and have coordinates. If the frog occupies the same coordinates are the lives, the user gets an extra life.*

Variables:

1. ints represent x and y coordinates
2. Boolean eaten: this represents whether or not the frog has eaten the specific life. Like enemies, it is easier to just not draw and ignore lives that are eaten than it is to actually remove them from the game.

Constructor:
Simply randomizes the the location of the lives it is creating

Methods:

I. update(): This does nothing but allows it to implement drawable
II. draw(Graphics g, String s): If the life is not eaten, this draws it

ObjectHolder: *This is our generic ObjectHolder which is used to store segments, obstacles, and lives. It is a LIFO queue in that the "end" is the newest object, but nothing is ever actually popped. It implements drawable, so everything it holds has draw and update methods. It is a simple linked list, and is based off of the one I made for labs 05 and 07. Its key functionality (aside from holding objects), is iterating through those objects and calling draw and update on*

all of them. It comes with a supporting class called GenericNode. An ObjectHolder list is made of GenericNodes which have an object and a pointer to the next Node.

Variables:

1. int length
2. GenericNode end: this is the end of the list

Constructor:
Sets the end Node as null and the length as 0.

Methods:

I. append(E toAppend): this method adds a new object to the ObjectHolder by wrapping it in a GenericNode and placing it at the end.
II. update(): calls the update method of the element stored in each node
III. draw(Graphics g, String biome): calls the draw method of the element stored in each node
    A. This and update() are called from other class's update methods
IV. locate(int index): returns the object of a given index. This is useful when, for example, in Frogger, you want to find and call methods on a specific segment in worlds' ObjectHolder of segments

Projectile: *Projectiles are the objects that the Frog shoots at the enemies. They are defined by positions and velocities.*

Variables:

1. doubles for x and y coordinates
2. doubles for x and y velocity

Constructor:
The constructor for Projectiles takes in a click location and the frog's current location then uses trigonometry to find the x and y velocities for a projectile moving between those two points. Speed is constant.

Methods:

I. update(): adds the x and y velocities to position
II. draw(Graphics g): draws the projectile

Abstract Class Segment: *Segments are what make up the map of the world. A world is divided into 15 of these, and they are either safeAreas or Roads. A frog is always "in" a segment (Visually and in terms of gameplay, but not in terms of the inner workings of the program). Segments implement drawable, so they have draw and update methods. This is an abstract class because you cannot have a segment that is not also one of its child classes. This needs to exist to allow Roads and SafeAreas to be stored in the same ObjectHolders*

Variables:

1. ObjectHolder obstacles: All segments have this ObjectHolder of obstacles
2. Public static final int height=50. All segments have the same height of 50
3. Int yCoordinate: Every Segment needs a y coordinate to be drawn in the right spot. They stretch across the entire screen, so they do not need an x coordinate.

Constructor: None

Methods:

I. draw ()
II. update(Graphics g, String biome)

SafeArea: *SafeArea extends segment and implements drawable, so it can be held in an ObjectHolder of segments. There are three safe areas in every world. SafeAreas are segments whose constructor does give them obstacles. The first, middle, and final segments are SafeAreas.*

Variables:

1. Static BufferedImage for cactus.png and snowman.png
2. Int array decorLocations: holds the integer x coordinates of the decor drawn on the safe areas

Constructor:
The constructor takes in and assigns a y coordinate to the SafeArea and randomizes and stores the locations of the decor (snowmen for tundra biome and cacti for desert). It does this in the constructor so that the locations are constant every time the safe area is drawn, rather than changing every frame.

Methods:

I. draw(Graphics g, String Biome): This draws the safe area. It takes in a string representing the biome and a Graphics object. Depending on the biome, it will draw either cacti or snowmen in random locations in the SafeArea for decoration.
II. update(): This does nothing, but allows it to implement drawable.

Road: *Roads are essentially the same as SafeAreas except that they have obstacles stored in an ObjectHolder.*

Variables:

1. ObjectHolder obstacles: this ObjectHolder is actually filled for Roads, unlike for SafeAreas.
2. int leftOrRight: all obstacles in a single road must move in the same direction. This integer represents that direction for each road
3. int laneMarkerStart: Simply the x coordinate where the dotted white line starts so its not entirely uniform

Constructor:
The Constructor for Road is more complicated than that for SafeArea. Its main function is creating the obstacles that it holds, adding them to the ObjectHolder spaced out relatively randomly and moving in the correct direction.

Methods:

I. update(): this method calls update on its obstacles ObjectHolder
II. draw(Graphics g, String biome): draws the background of the road and calls draw on its obstacles ObjectHolder

Abstract Class Obstacle, Classes Sedan and Truck: *Obstacle is an abstract class representing the vehicles that occupy the roads. Obstacles are defined by a pair of coordinates and a velocity. Its two child classes are* ***Sedan*** *and* ***Truck****. The only actual difference between the two children is their size and the image that gets drawn to represent them on the screen.*

Variables:

1. Ints for x and y coordinates and velocity
2. Its children, Sedan and Truck, each have static images that are their representations on the screen. These are static, so they only have to be read into java once-- at the beginning of the game-- rather than every time we draw the vehicles, which noticeably slowed down the game.

Constructor:
There is no constructor for general obstacles, but the constructor for each of its child classes simply sets the length of the object and the direction of its velocity based on the information passed in.

Methods:

I. update(): This is contained in **Obstacle** and simply adds velocity to x position.
II. draw(Graphics g, String biome): This is specific to each child class. It draws the subclass-specific image of the vehicle to the screen at the correct coordinates
III. readImage() (one for each subclass): these static methods are called before the game starts to read in the images. It is static because we have it reading in the images and storing them as static variables before instances of the class even exist. Reading the image every time the vehicles are drawn slowed down the game too much, and this allows the method to only ever need to be called once.

Interface drawable: *The reason this interface exists is to standardize all objects held in ObjectHolders to ensure that they are compatible with its draw and update methods.*

Methods:

I. draw(Graphics g, String biome)
II. update()

How the Classes Interact:

The four main functions of our program are updating locations of objects, drawing objects to the screen, tracking temperature of the frog, and checking for collisions. Calls to update methods originate in the main loop of the game in Frogger and eventually end up calling the update methods held in each class. These calls do often go through intermediate classes (e.g. Frogger calls world.updateSegments() and the method in world then calls update on the segments). Drawing similarly originates in Frogger, but the draw methods are unique to classes. Temperature mechanics methods are in Frog, but the calls to them are in Frogger. Finally, collisions checking usually occurs in the World or Frogger because these are the classes whose scopes allow them access to the needed objects (e.g. World contains both the Projectiles and the Enemies).