EECS 2030 Project:  
  
Snake World

Jason Skinner, Alessa Ivascu, Dawood Choksi, Leroy Nguyen, Kristiana Papajani

Description of Code

* The snake game project consists of many classes that are blueprints for its objects.
* The project consist of the main classes are MVC (**Model**.java, **Controller.**java**,** and **View**.java), which is a software architectural pattern for implementing user interfaces on computers.
* There are other classes that take care the objects of the game which are: **AudioFilePlayer.java**, **HS.java**, **RowCol.java**, and **Snake.java**. Lastly, in order to run and execute the snake game, requires a PSVM which is found in Game.java. These classes all adhere to the MVC which embodies the objects of the Snake Game.
* The **Model** separates the logic of a program from the rest of the user interface. It manages the data and fundamental behaviours of a program. It is the data and a data-management portion of the program.
  + In our Model class, we have *constant* fields of numbers and strings which are public, private, static, and final.
  + We have an empty Mode class constructor which adheres to the MVC pattern.
  + A showHighScores() is constructed, where the Model retrieves the high scores from the HS class and then displays it them into in a pop-up message dialog window.
    - In order to implement the method, requires a for loop technique.
  + The game includes a multiplayer mode, where it introduces a second player on the left-hand side of the Graphic User Interface (GUI), whereas the first player is on the right-side. Thus the new game is set up for both players.
    - The newTwoPlayerGame() method is responsible for this.
  + The model takes care of the image of the snake as it moves across the game board for both single and multiplayer, as it moves in the game board or background of the GUI. For example, as the snake head moves in a direction forward, the body of the snake follows the previous position of what the snake head was just in. There are two types of images for the snake’s body and four types for the snakes head that must be considered depending on the DIRECTION (up, down, right, left) of the snake’s movement.
    - Thus, changeHeadToBody(Snake snake) is implemented for this behaviour.
    - The method showSnakeBody(Snake snake) takes care of the movement of the snake’s body as it moves around at any direction.
* At the very start of the game after running Game.java, the player will be immediately prompted and be asked to optionally if he/she wants to play the background music, and as well as choosing their player's usernames. Then after, the game refreshes and sets up for the player(s).
* This is evident in setView(View view) method.

* + When the snake dies in the game, the game immediately stops and resets when the user wants to try and play again.
    - So the Model calls the showSnakeGone(Snake snake) and erases the snake in the game.
  + So depending on which direction of the snake is moving, it will change it to either left, right, upward or downward direction.
    - The methods: leftPressed(String snake), rightPressed(String snake), upPressed(String snake), and downPressed(String snake) is manages the movements of the player(s) snakes.

* In the Snake class, we have used an encapsulation where we have all our fields/attributes in **private** (i.e; private int length, private RowCol head, private ArrayList<RowCol>…etc.) and getters which are being accessed by the Model class.

Application of Techniques Learned

The following techniques are used in multiple locations throughout the project’s code, however, only some specific instances are highlighted for the purposes of the project’s requirements.

All 12 **Basic Techniques** are used in the project, as well as all the **Advanced Techniques**.

# Encapsulation / Information Hiding:

* Used throughout all classes in the project.
* For example, the class Public Static HighScores class inside the Model class; the fields are marked as private, as well as every constructor and non-gettor method.
* The code in this class is therefore hidden and cannot be accessed randomly.

# Overloading / Constructors:

* Overloading occurs in many places in the code throughout this project.
* For example, in the RowCol class, the constructors are overloaded.
* One takes in an integer value for the number of rows and columns and creates an object, setting the objects number of rows and columns according to the numbers inputted.
* The second constructor for the RowCol class takes in another RowCol object as a parameter and copies its number of rows and columns and creates a new object using those numbers to set the number of rows and columns.
* An example of overloaded methods is in the Snake class, and specifically the checkForSnake methods.
* One method accepts a single snake object as a parameter, representing a player, and checks to see if the snake collides with itself.
* The other method of checkForSnake accepts 2 snake objects for parameters, each representing different players, and checks to see if the snakes collide with each other.

# Static Methods and Static Variables:

* Static Methods and Static Variables are used throughout the project, namely to keep a counter or instance of a single object, or for allowing certain methods to not require an object to perform their function.
* The class Public Static HighScores inside the Model Class is an example of a class using static methods and static variables.
* All the fields are static, such as the field maintaining the highscores.txt file, as only 1 instance is needed for all objects of HighScores.
* The methods are static and belong to the HighScores class, and do not act on objects/instances of HighScore.

# Mutable and Immutable Classes:

* Most classes in this project are mutable for the purposes of functionality.
* An example of a mutable class is the Snake class. The methods and constructors are all Public, and specifically the settors are public as well, thus Snake objects can have its values changed. The Snake’s length and initial head position settors are Public, and as such, can be changed to different lengths or initial positions.
* An example of an immutable class is the class Public Static HighScores inside the Model class; all fields are marked private and most methods are private, including the settors. Only the gettor is public, and since it is coded properly, it will not have access to HighScores objects. The class itself is also static and thusly independent of the Model class.

# Inner Classes:

* Inner classes are used throughout the project, though they may not be widely noticeable as they’re usually anonymous classes inside of other classes.
* For example, in the Model class, inside the newGame method, there is a runnable stream that creates an anonymous inner class for the Runnable object to create the timer stream.
* A more apparent and obvious inner class is the public Static HighScores class inside the Model class.

# Interfaces/Abstract Classes:

* Interfaces are used in a few locations in this project.
* One of the key uses of interfaces is in the Controller class.
* The Controller class implements the ActionListener and KeyListener interfaces and their associated methods.
* The ActionListener interface allows for methods that “listen” for user inputs such as mouse clicks, and thus those clicks can then be used to perform something, such as clicking on a menu option would open a dialog box.
* The KeyListener interface allows for methods to “listen” for user inputs from the keyboard, and thus those key presses can be used as parameters for methods to perform a function when a key is pressed, such as moving the snake for a single player or 2 players depending on the key pressed.

# Inheritance:

* A crucial example of inheritance in this project is in the View.
* The public class View inherits from the JFrame class; all of JFrame’s methods and attributes are available to the View and thusly to the project.
* Many of the things in JFrame are needed for Swing and to construct valuable features of the Snake game.
* For example, the entire Snake game window is constructed using various JFrame methods like setText and setting dimensions of the window itself. Other things like setting the Menu bar using JMenuBar and JMenuItems lets us implement menus for the Snake game. Even simple things like JFrame.EXIT\_ON\_CLOSE lets the application terminate when the application is closed, like when the “X” in the top right of the game’s window is clicked.

# Polymorphism:

* Polymorphism is used in many places throughout this project.
* A simple example of the use of polymorphism is in the RowCol class.
* At the very end of the class, there is a method named public String toString(). Since a toString() method is defined, the objects of RowCol use this toString() method, and not the toString() method from the Object class. This occurs due to Late Binding.

# Generics:

* Generics are used throughout the project in terms of ArrayLists. However, a specific example of a Generic class is the HS class.
* This is a generic class with 2 type parameters. All the constructors and methods are coded in a way such that the 2 types, S and N, can be any type.
* In the Model, the types for this Generic class is then defined as needed. In our project, we require the high scores to be of the format where the first type is a String and the second type is an Integer.
* S and N are used as the names of the types to provide some meaning to the variable names; the S represents a shortened form of String, while N represents a shortened form of Number. It is important to note these are not always String and numbers/Integers, as the class is a generic, but these variable names help remember the order that a high score should appear in (String, Number/Integer).

# Swing/GUI/Event-Driven Programming:

* Swing/GUI is used to make the actual window and display the game.
* All of the “drawing” of the game is handled in the class called View.
* For example, the constructor of the view uses many Swing/GUI coding commands like making the window itself, making the window resizable, setting the initial background (which is later changed), setting the Snake image, making menus, packing it all together, etc.
* The key part of the View is the Event-Driven programming which occurs in the constructor which accepts parameters of type ActionListener and KeyListener.
* The ActionListener allows for methods that “listen” for user inputs such as mouse clicks, and thus those clicks can then be used to perform something, such as clicking on a menu option would open a dialog box.
* The KeyListener allows for methods to “listen” for user inputs from the keyboard, and thus those key presses can be used as parameters for methods to perform a function when a key is pressed, such as moving the snake for a single player or 2 players depending on the key pressed.
* There are methods that also accept these Listeners are parameters, such as the makeMenu method which “listens” for an action such as a mouse click and then generates the appropriate menu depending on which button is clicked.
* The advantage of using ActionListener and KeyListener is that it helps with Event-Driven Programming; the methods that listen for these actions and key presses only execute when an action or key press is detected, and thus are not performing all the time.

# Array and ArrayList Collections:

* ArrayList collections are used throughout the project.
* An example of an ArrayList is in the Snake class, where the private attribute ‘body’ is an ArrayList Collection.
* This is useful as the ‘body’ ArrayList keeps track of the number of body elements of the snake; as the snake eats food units, the ‘body’ ArrayList increases in length between 1-3 units randomly. Therefore, an ArrayList, and not a simple array with a defined length, is useful here.
* Another example of an ArrayList is in the HighScores inner class of the Model class. The HighScores class has an ArrayList called ‘scores’ that is used to contain all the high scores.
* Arrays are seldom used in our project. A simple example is in the View, in the randomFood method. Here, a simple Array containing the ‘x, y’ position is used for the food unit’s placement. The array is a simple 2D array and its elements are calculated randomly using the game’s width and height.

# Exceptions and File I/O:

* Exceptions are used throughout our project.
* An example of an exception used multiple times is the InterruptedException, which is used when a thread is interrupted rather put to sleep.
* This occurs in our Model many times, but a key place is in the newGame method, where the Timer stream is coded. A Try/Catch block is used around the code of the Timer, and if the timer thread is interrupted instead of sleeping, the Catch block will catch the InterruptedException and alert the user.
* A Throwable exception is used in the last method, getLines, of the inner class HighScores inside the Model class. The method will throw an IOException to the user if there is a problem with the input/output in the reading/returning of highscores.
* In the inner class HighScores inside the Model Class, the high scores are stored in an ArrayList and written to a text file. The high scores are also read off the text file and shown to the user. This all occurs in the writeHighScores method, getHighScores method, and getLines method.

Application of ADVANCED Techniques Learned

In this section, most of the advanced techniques and how they’re used in the code of this project is discussed.

# Recursion:

* An example of Recursion is in the public static HighScores class inside the Model class. The very last method, getLines(), functions recursively in its execution.
* The getLines() method reads the lines of a file one by one, and adds them to an ArrayList. When there are no lines left to read, the method returns the ArrayList.
* This follows the guidelines for proper use of Recursion:
  + The method getLines calls itself.
  + The method getLines has a base case; the base case is 0 lines left to read.
  + The method getLines is always getting closer to the base/shrinking; the number of lines to read starts at the max and gets to 0 eventually.

# Model/View/Controller:

* The Model View Controller classes are used in this project.
* The Model handles the majority of “data” or “functionality” code, such as making a new game and its timer, handling high scores, handling multiplayer, etc.
* The View handles the “drawing” of the Snake game and the Swing/GUI elements.
* The Controller connects the Model and View together. Things like key presses are handled in here, where the key press is detected by the View and “sent” to the model to update the Snake on what should happen after the key press, and then “sent” back to the view to update the Snake/game to show what happens after the key is pressed.

# Functional Programming:

# Multithreading:

* Multithreading is used in a few places in the code for the project.
* An example of multithreading is used in the coding of the Timer. This happens in the Model class, in the newGame() method.
* A new thread is created after the new Runnable is created, which is called ‘time’. Thus, a 2nd thread is created here to handle the Timer shown in the Snake game window.

# Networking and Sockets:

* The Networking for Multiplayer is handled in the Networking class.
* First a stream for the Networking portion is created.
* Then a network listener is created by making a new Runnable object. In here, a new serverSocket object is created, which listens to an incoming client/other player. A clientSocket is also created, which “talks”/sends information from the client.
* All the received information from the sockets is converted into a string.
* This string output is then sent to the browser/displayed to the user.
* Afterward, all the open sockets are closed, ex. Web streams, client and server sockets.
* All of the networking information, aka web input/output, is logged to a text file.
* The networking is all done in a separate multithread.