**EECS 2030 Final Project**

Snake Game

Haider Al-Tahan  
213788708

Jianxiong Wang

Ammiel Cruz

Kamil Sarbinowski

**Project Description:**

This project is a custom swing based snake game. The objective of the game is to eat as many apples as possible. Every time the snake eats an apple, its body grows. There are three types of apples in this game, the regular Apple, Golden Apple, and Poisoned Apple. Eating the Golden Apple increases the size of the snake by 2 and the score of the player by 50. On the other hand, the regular Apple increases the size of the snake by 1 unit and score of the player by 1-3 randomly. Finally, consuming the Poisoned Apple ends the game. The game also ends if the player hits any of the four boundaries or the snake itself.

The game implements four difficulty levels: SLOW, MEDIUM, FAST, and EXTREME. Every difficulty level has different speed, level length, maximum numbers of Apples, and score multiplier. Whenever the snake finishes the current difficulty level, the game is restarted with next difficulty level and the score is accumulated. As the difficulty increases, the snake moves faster, and more Apples appear on the screen. The worth of both the regular Apple and the Golden Apple increases based on the current difficulty score multiplier. Finally, if the snake completes the last difficulty level, the difficulty resets to LOW.

The snake game store and sync the name and score of the player upon losing the game to a NoSQL Database. Data is synced across all clients in real-time, and remains available when the application goes offline. This was made possible by using a google service called Firebase Real-time Database. The Firebase Real-time Database is a cloud-hosted database where data is stored as JSON and synchronized in real-time to every connected client. Additionally, all of the clients share one Real-time Database instance and automatically receive updates with the newest data.

**Features and implementations:**

The project demonstrates a number of techniques learned during the course. These techniques include encapsulation/information hiding, overloading/constructors, static methods and static variables, mutable and immutable classes, inner classes, interfaces/abstract classes, inheritance, polymorphism, generics, swing, array and arrayList/Collections, exceptions and file I/O, multithreading, and networking and sockets. These techniques are covered in detail in the following paragraphs.

**Encapsulation/Information Hiding**

Information Hiding and Encapsulation are complementary concepts. Information Hiding is the practice of separating how to use a class from the details of its implementation, i.e. a mechanism for restricting access to some of the object’s components. On the other hand, Encapsulation means that the data and methods of a class are combined into a single unit, which hides the implementation details (JAVA provides encapsulation using class). In this project, Encapsulation was used by facilitating a bundle of data with the methods operating on data by using Classes. Additionally, Information Hiding was used for restricting access to some of the object’s components by using the private modifier. For example, the Tile class bundle data responsible for coordination and holds private x and y variables to restrict its access.

**Overloading/Constructors**

Constructors in a class are invoked to initialize the object. Overloading is the ability to create multiple methods/constructors of the same name with different implementations. In this project, overloading and constructors were used both at the same time to create multiple constructors for different uses. For example, the Tile class uses these concepts by implementing a default constructor and a copy constructor as seen in the figure below.

*/\*\*  
 \* Constructor  
 \*  
 \** ***@param*** *x x coordinate  
 \** ***@param*** *y y coordinate  
 \*/*public Tile(int x, int y) {  
 this.x = x;  
 this.y = y;  
}  
  
 */\*\*  
 \* Copy Constructor  
 \*  
 \** ***@param*** *t another Tile  
 \*/*public Tile(Tile t) {  
 this.x = t.x;  
 this.y = t.y;  
 }

**Static Methods and Variables**

Static methods and variables belongs to the class and not to the object (instance) and can access only static data, it cannot access non-static data (instance variables). In this project, the static modifier is used mainly in the Constants class. Therefore, there is no need to initiate the Constants class to use the variables. A separate class was created that contained constants to which classes were able to reference from.

public final class Constants {  
  
 public static final String *VERSION* = "v1.0.0";  
 public static final String *GAME\_TITLE* = "Snake Game - EECS 2030 Project - " + *VERSION* ;  
 public static final String *FIREBASE\_FILE\_PATH* = "service-account.json";  
 public static final String *FIREBASE\_LINK* = "https://snakegame-2a153.firebaseio.com/";  
 public static final String *DATABASE\_MAIN\_OBJECT* = "Scores";  
 public static final String *DATABASE\_ERROR\_OBJECT* = "Error";  
  
 public static final int *HEIGHT* = 625;  
 public static final int *WIDTH* = 800;  
 public static final int *GAME\_WIDTH* = 600;  
 public static final int *GAME\_HEIGHT* = 600;  
 public static final int *DOT\_SIZE* = 25;  
  
 public static final String *START\_COMMAND* = "start";  
 public static final String *EXIT\_COMMAND* = "exit";  
  
 public static final int *HORIZONTAL\_PADDING* = 10;  
 public static final int *VERTICAL\_PADDING* = 25;  
  
 public static final String *HIGHSCORES\_HEADER\_NAME* = "Player Name";  
 public static final String *HIGHSCORES\_HEADER\_POINTS* = "Points";

**Mutable and Immutable Classes**

Mutable classes have fields and state that can be changed, while immutable classes are the opposite. StringBuilder, java.util.Date are a few examples that are mutable. String and boxed primitives like Integer and Long cannot be changed. Within the game, mutable models were created to be able to change their state. For example, the Score class has getters and constructors to be able to access and set its fields.

public final class Score implements Comparable<Score> {  
  
 private final String name;  
 private final int points;  
  
 */\*\*  
 \* Constructor  
 \*  
 \** ***@param*** *name The name of the player  
 \** ***@param*** *points The points of the player  
 \*/* public Score(String name, int points) {  
 this.name = new String(name);  
 this.points = new Integer(points);  
 }  
  
 */\*\*  
 \* Constructor  
 \*  
 \** ***@param*** *jsonSRC Parses the JSON representation of the player and points  
 \*/* public Score(String jsonSRC) {  
 this.name = jsonSRC.substring(jsonSRC.indexOf("name=") + 5, jsonSRC.indexOf(','));  
 this.points = Integer.*parseInt*(jsonSRC.substring(jsonSRC.indexOf("points=") + 7, jsonSRC.indexOf('}')));  
 }  
  
 */\*\*  
 \* Returns the current player's name  
 \*  
 \** ***@return*** *The current player's name  
 \*/* public String getName() {  
 return this.name;  
 }

**Inner Classes**

An Inner class is a class that has another class declared inside the main class or interface. This is done to logically group classes and interfaces in one place so that it can be more readable and maintainable. In addition, it the inner class can access all of the members of the outer class, including private data members and methods. Inner classes were implemented within the GameController java file. The class TAdapter is an inner class of the outer class GameController.

public class GameController extends JPanel implements ActionListener, Runnable{  
  
 private GameView gameView;  
 private GameStatusBar gameStatusBar;  
 private GameModel gameModel;  
  
 private Difficulty difficulty = Difficulty.*SLOW*; // delay for timer  
 private int levelLength = difficulty.getLevelLength(); // the snake length to upgrade the game difficulty level  
 private Timer timer;  
public GameController(String playerName) {  
 super(new BorderLayout());  
 addKeyListener(new TAdapter());  
 this.gameModel = new GameModel(playerName);  
 this.gameStatusBar = new GameStatusBar(playerName);  
 this.gameView = new GameView(this.gameModel);  
 }  
  
 …GameControllerMethods…  
private class TAdapter extends KeyAdapter {  
  
 @Override  
 public void keyPressed(KeyEvent e) {  
 int key = e.getKeyCode();  
 if (key == KeyEvent.*VK\_R*) {  
 gameModel.initGame(difficulty);  
 resetTimer();  
 }  
 else if (key == KeyEvent.*VK\_Q*) System.*exit*(0);  
 else gameModel.setDirection(key);  
 }  
 }  
}

**Interfaces / Abstract Classes**

Abstract classes were used in this project to provide some default behavior for any class that extends Buffer. In particular, the subclasses Apple, PoisonedApple, and GoldenApple must implement an addTo method within each class.

public abstract class Buffer extends Tile {  
  
 */\*\*  
 \* Constructor  
 \*  
 \** ***@param*** *x x coordinate  
 \** ***@param*** *y y coordinate  
 \*/* public Buffer(int x, int y) {  
 super(x,y);  
 }  
  
  
 */\*\*  
 \* A copy constructor  
 \*  
 \** ***@param*** *t a Tile  
 \*/* public Buffer(Tile t) {  
 this(t.getX(), t.getY());  
 }  
  
 */\*\*  
 \* Add this buffer to a snake.  
 \*  
 \** ***@param*** *snake the snake  
 \*/* public abstract void addTo(Snake snake, Difficulty difficulty);  
  
}

**Inheritance**

Objected-oriented programming allows classes to inherit commonly used state and behavior from other classes. In the example below, The Buffer class becomes the superclass of Apple. In Java, each class is allowed to have one direct superclass, and each superclass has the potential for an unlimited number of subclasses.

public final class Apple extends Buffer {  
  
 */\*\*  
 \* Constructor  
 \*  
 \** ***@param*** *x x coordinate  
 \** ***@param*** *y y coordinate  
 \*/* public Apple(int x, int y) {  
 super(x, y);  
 }  
  
 */\*\*  
 \* Copy constructor  
 \*  
 \** ***@param*** *t a tile  
 \*/* public Apple(Tile t) {  
 super(t);  
 }

**Polymorphism**

Polymorphism is the ability for an object to take many forms. In Java, any object that can pass the IS-A test is considered to be polymorphic. This is because any object will pass the IS-A test for their own type and for the class Object.

**Generics**

Generics enable types to be parametesr when defining classes, interfaces, and methods. This allows for the same code to be reused with different inputs. The use of generics in the snake game can be found in the GameModel class where List<T> buffers = new ArrayList<T>(); is used. In this case, the type Buffer is substituted for T as seen below.

public class GameModel {  
  
 private final String playerName;  
 private Difficulty difficulty; // number of maximum buffers that can be present on the board  
  
 private Snake snake;  
 private List<Buffer> buffers = new ArrayList<>();  
 private Class<?>[] bufferTypes = new Class[]{GoldenApple.class, PoisonedApple.class};  
 private boolean ableToSetDirection = true;  
 private int cycleCounter = 0; // number of game cycles  
 private boolean inGame = true;

**Swing / GUI / Event-Driven Programming**

Swing is a GUI widget toolkit for Java and was used in this project. The swing library makes use of the Model/View/Controller design pattern, which allows for separating the Model containing the logic with the View. The Controler acts as the bridge between the two.

**Array and ArrayList / Collections**

The ArrayList class extends the AbstractList and implements the List interface. Unlike standard Java arrays that are of a fixed length, ArrayList supports dynamic arrays that can grow and are created with an initial size. When this size is exceeded, the collection is automatically enlarged. The array may be shrunk when objects are removed. ArrayList is used in the GameModel class where an ArrayList is created that holds Buffer objects.

public class GameModel {  
  
 private final String playerName;  
 private Difficulty difficulty; // number of maximum buffers that can be present on the board  
  
 private Snake snake;  
 private List<Buffer> buffers = new ArrayList<>();  
 private Class<?>[] bufferTypes = new Class[]{GoldenApple.class, PoisonedApple.class};  
 private boolean ableToSetDirection = true;  
 private int cycleCounter = 0; // number of game cycles  
 private boolean inGame = true;

**Exceptions and File I/O**

Input/output in Java is used to process the input and produce the output. Java File class represents the files and directory pathnames in an abstract manner. This class is used for creation of files and directories, file searching and file deletion among other things.

public final class Database{  
  
 private FirebaseDatabase database;  
 private DatabaseReference databaseRef;  
 private DatabaseReference databaseRefError;  
   
 private static Database *INSTANCE*;  
  
 private Database() throws Exception {  
 File file = new File(this.getClass().getResource(Constants.*FIREBASE\_FILE\_PATH*).getPath());  
 FileInputStream serviceAccount = new FileInputStream(file);  
  
 FirebaseOptions options = new FirebaseOptions.Builder()  
 .setCredential(FirebaseCredentials.*fromCertificate*(serviceAccount))  
 .setDatabaseUrl(Constants.*FIREBASE\_LINK*)  
 .build();  
  
 FirebaseApp.*initializeApp*(options);  
  
 this.database = FirebaseDatabase.*getInstance*();  
 this.databaseRef = this.database.getReference(Constants.*DATABASE\_MAIN\_OBJECT*);  
 this.databaseRefError = this.database.getReference(Constants.*DATABASE\_ERROR\_OBJECT*);  
 }

**Multithreading**

Multithreading is a process of executing multiple threads simultaneously. Threading is a lightweight sub-process and the smallest unit of processing. Threads share a common memory area and don’t allocate separate memory area, which saves memory. Context-switching between the threads take less time as well. Advantages include non-blocking the user since threads are independent and multiple operations can be performed at the same time. This results in saved time. Threads running independently also do not affect other threads if exception occurs in a single thread. Finally, cost of communication between threads is low.

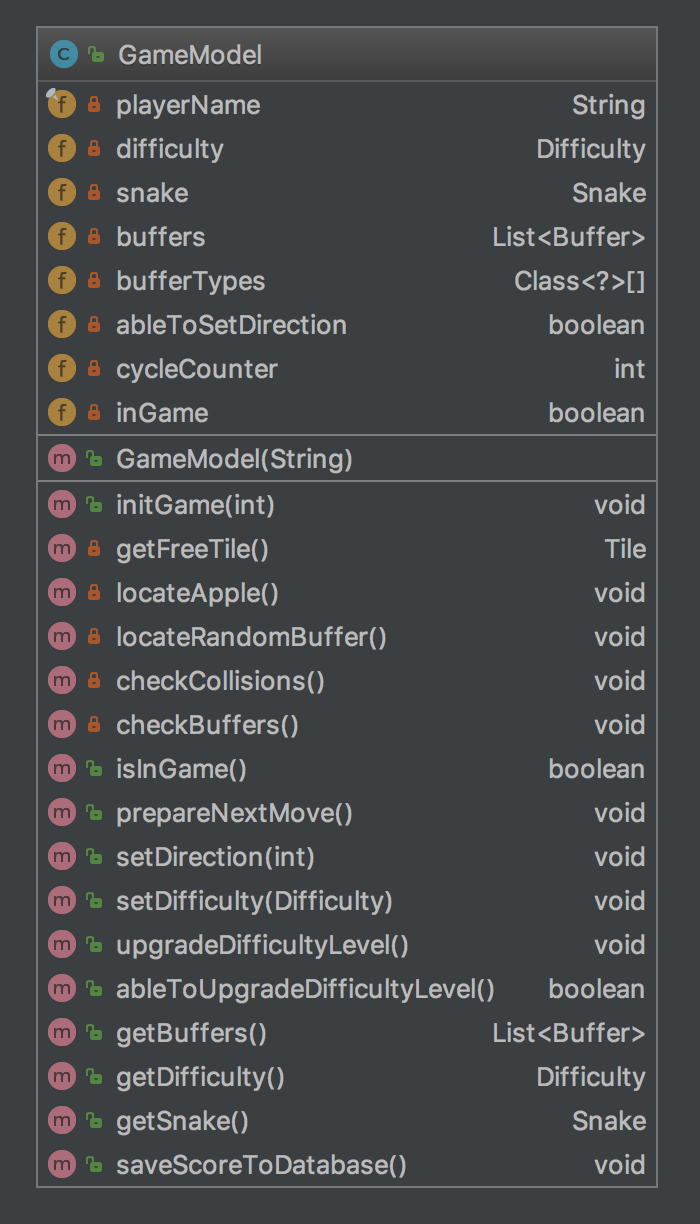
In this project, multithreading was applied through the Runnable Interface that the GameController implements.

**Junit:**

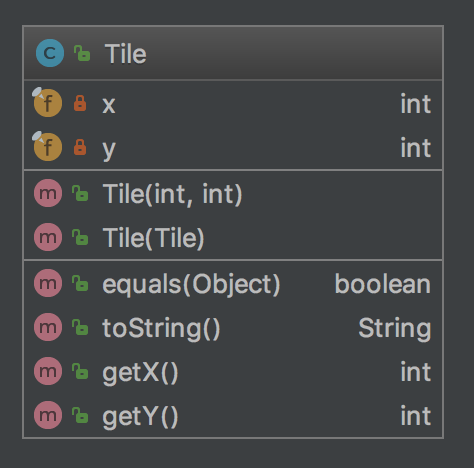
Unit testing is a software development process in which the smallest testable parts of an application, called units, are tested separately for proper operation. In this project, the JUnit4 framework is used to test select models. Various test cases were created for select methods in crucial model classes. Only certain cases were considered since it was impractical to test all possible sets of arguments. Arguments that had typical values and that tested boundary cases were only considered.

**Select Class Diagrams:**

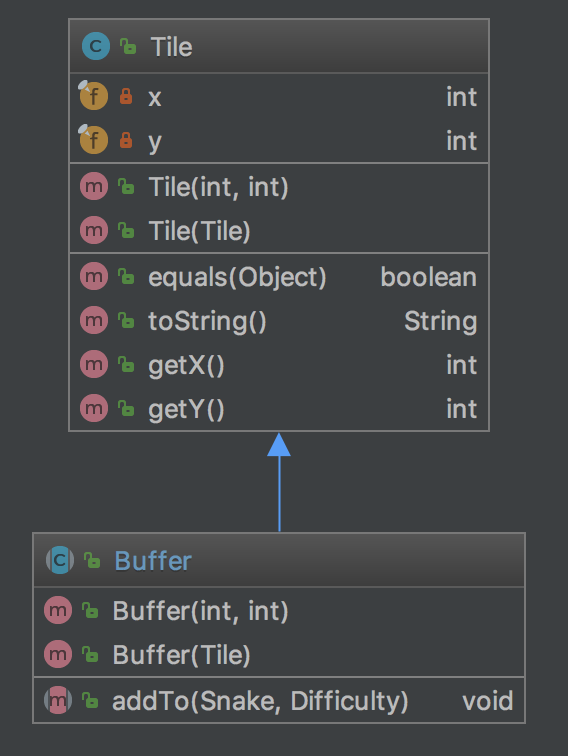
**GameModel**

****

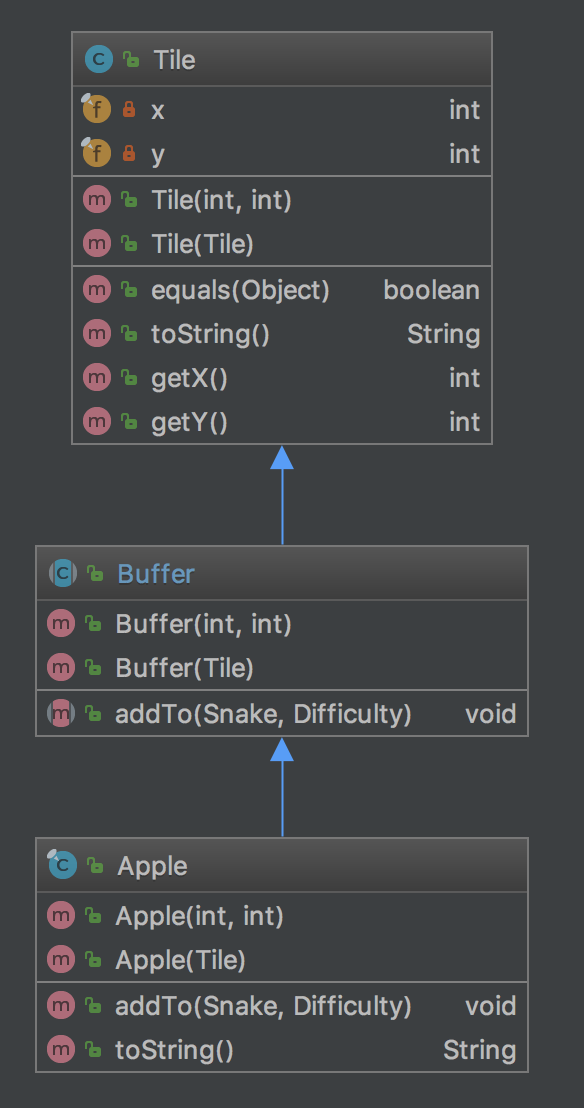
**Tile**

****

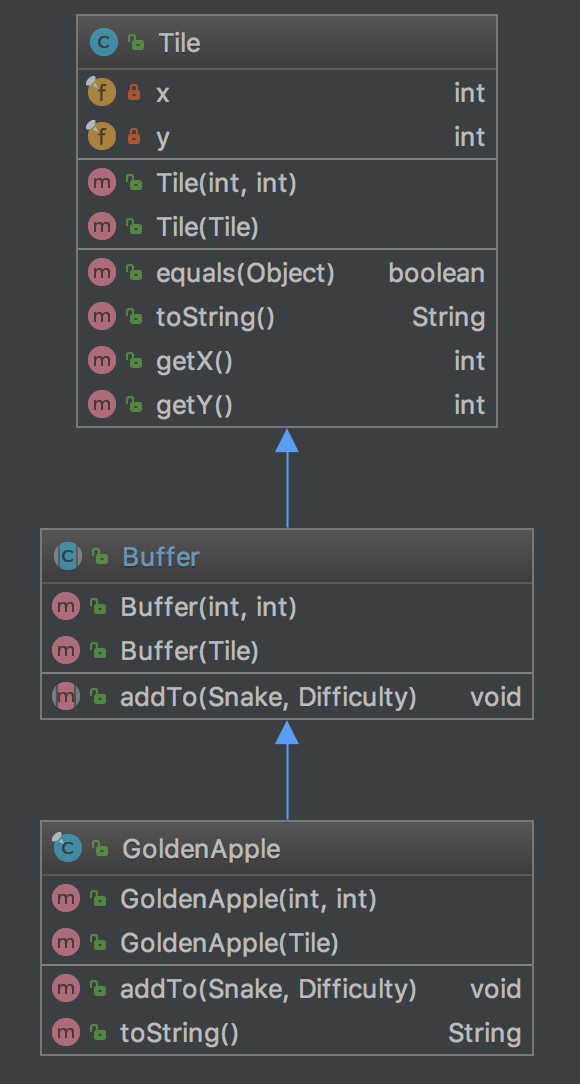
**Buffer**

****

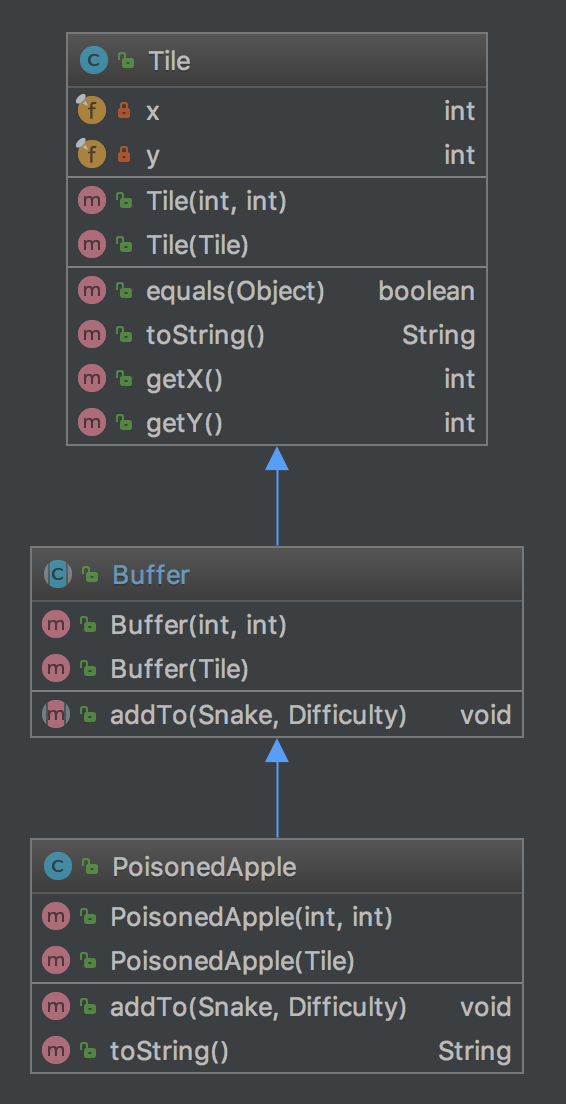
**Apple**

****

**GoldenApple**

****

**PoisonApple**

****