EECS 2030 Final Project

Snake Game

Haider Al-Tahan
Jianxiong Wang
Ammiel Cruz
Kamil Sarbinowski

Project Description

This project is a custom Swing-based snake game. The player plays as a snake and 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 two and the score of the player by fifty. On the other hand, the regular Apple increases the size of the snake by one unit and the score of the player by 1-3. Finally, consuming the Poisoned Apple, hitting any of the four boundaries, or the snake itself ends the game.

The game implements four modes of difficulty: SLOW, MEDIUM, FAST, and EXTREME. Every difficulty level has a different speed, level length, maximum number of Apples, and score multiplier. Whenever the snake finishes the current difficulty level, the game restarts to the next difficulty level with the score 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 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 a Real-time Database instance and automatically receive updates with the newest data.

Features and implementations

The project demonstrates several 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 while overloading is the ability to create multiple methods/constructors of the same name with different implementations. In this project, overloading and constructors were both used at the same time to create multiple constructors for different uses. Going back to the Tile class example, it uses these concepts by implementing a default constructor and a copy constructor as seen in the figure below. The keyword @Override allowed for custom equals and toString methods for the Tile class.

```
public class Tile {
  private final int x;
  private final int y;

  public Tile(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public Tile(Tile t) {
     this.x = t.x;
     this.y = t.y;
   }

  @Override
  public boolean equals(Object obj) {
    if (this == obj) return true;
    if (obj == null || !(obj instanceof Tile)) return false;
    Tile other = (Tile) obj;
    return this.x == other.x && this.y == other.y;
  }

  @Override
  public String toString() {
    return String.format("Tile x=%d y=%d",x,y);
  }

  public int getX() {
    return x;
  }

  public int getY() {
    return 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). The static modifier was used in this project, mainly in the Constants class. This allowed classes to reference this class for constant variables and allow for code re-use.

```java
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;

   …
```

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 while 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. An example of a mutable class is the Snake class where it has setters and getters that could access and modify its fields.

```java
public class Snake {
  private final LinkedList<Tile> tiles = new LinkedList<Tile>();
  private Directions direction = Directions.EAST;
  private int length = 3;
  private int remainGains = 0;

  private int score = 0;
  private boolean isAlive = true;

  public Snake() {
    this(0);
  }

  public Snake(int score) {
     this.score = score;
      direction = Directions.EAST;
      length = 3;
      int xOffsetUnits = 2;
      int yOffset = Constants.DOT_SIZE * 3;
      for (int i=0; i<length; i++)
         tiles.add(new Tile(Constants.DOT_SIZE*(length-i-1 +xOffsetUnits), yOffset));
```

```java
}

public void move() {
  Tile newHead = tiles.getFirst();
  if (this.remainGains == 0) {
    tiles.removeLast();
  } else {
    this.remainGains--;
  }
  switch (direction) {
  case WEST: newHead = new Tile(newHead.getX()-Constants.DOT_SIZE, newHead.getY()); break;
  case EAST: newHead = new Tile(newHead.getX()+Constants.DOT_SIZE, newHead.getY()); break;
  case NORTH: newHead = new Tile(newHead.getX(), newHead.getY()-Constants.DOT_SIZE); break;
  case SOUTH: newHead = new Tile(newHead.getX(), newHead.getY()+Constants.DOT_SIZE); break;
  }
  tiles.addFirst(newHead);
}

public boolean containsTile(Tile t) {
  return this.tiles.contains(t);
}

public boolean isAlive() {
   if (tiles.lastIndexOf(tiles.getFirst()) > 0)
      this.isAlive = false;
   return this.isAlive;
}

public Directions getDirection() {
  return this.direction;
}

public Tile getHead() {
  return this.tiles.getFirst();
}

public int getScore() {
  return this.score;
}

public int getLength() { return this.tiles.size(); }

public Iterator<Tile> getBodyIterator() {
  Iterator<Tile> iter = this.tiles.iterator();
  iter.next();
  return iter;
}

public void setAlive(boolean a) {
   this.isAlive = a;
}

public void setDirection(Directions d) {
  this.direction = d;
}

public void gains(int n) {
  this.remainGains += n;
}

public void addScore(int n) {
  this.score += n;
```

```
    }

    public void addBuffer(Buffer b, Difficulty difficulty) {
        b.addTo(this, difficulty);
    }
}
```

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, the inner class can access all the members of the outer class, including private data members and methods. Inner classes were implemented within the GameController java file. In the example below, the class TAdapter is an inner class of the outer class GameController and is able to access the outer class's (GameController) private fields.

```
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

An abstract class is a class that is declared abstract and it may or may not include abstract methods. They are like interfaces; however, you can declare fields that are not static and final, and define public, protected, and private concrete methods. Abstract classes were used in this project to provide some default behavior for classes that extends the Buffer class. The subclasses Apple, PoisonedApple, and GoldenApple required the addTo method to be implemented within each class as seen below.

```
public abstract class Buffer extends Tile {

    public Buffer(int x, int y) {
        super(x,y);
    }

    public Buffer(Tile t) {
        this(t.getX(), t.getY());
    }

    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 Java, each class can have one direct superclass, and each superclass has the potential for an unlimited number of subclasses. The Buffer class became the superclass of Apple as seen in the example below.

```
public final class Apple extends Buffer {

    public Apple(int x, int y) {
        super(x, y);
    }

    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. Classes that extended another class in the project were polymorphic by default.

Generics

Generics enable types to be parameters 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;
   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

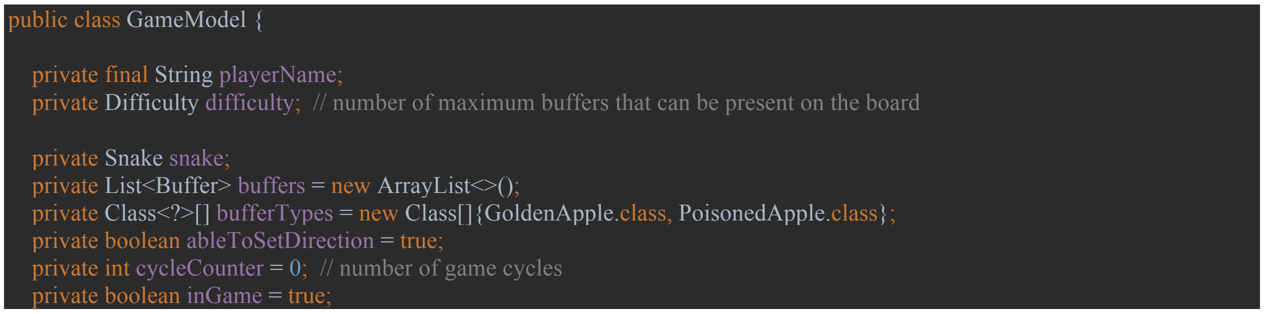
Swing is a GUI widget toolkit for Java and provides the ability to create graphical user interface components, such as buttons, a window, and scroll bar to name a few. The framework provides a layer of abstraction between the code structure and graphic presentation of a Swing-based GUI. It is highly modular, allowing users to provide their own custom implementations of these components using Java's inheritance mechanism. The swing library also makes use of the Model/View/Controller design pattern, which allows for separation between the Model containing the logic, and the View. The Controller acts as the bridge between the two.

MVC Pattern

The MVC design pattern is the separation of input, processing, and output of an application. This model divides into three interconnected parts called the Model, the View, and the Controller. This project utilized this model because of a few important advantages it brought in development. The first one is faster development process, allowing multiple programmers to work on one of the modules at a time. The second one is less breaking changes when modifications in one part of the modules are made. Changes do not affect the entire thing. Finally, the MVC model allows support for asynchronous techniques. The project was structured such that the model classes were placed under the Models folder, while the controllers were placed in the root directory.

Array and ArrayList /Collections

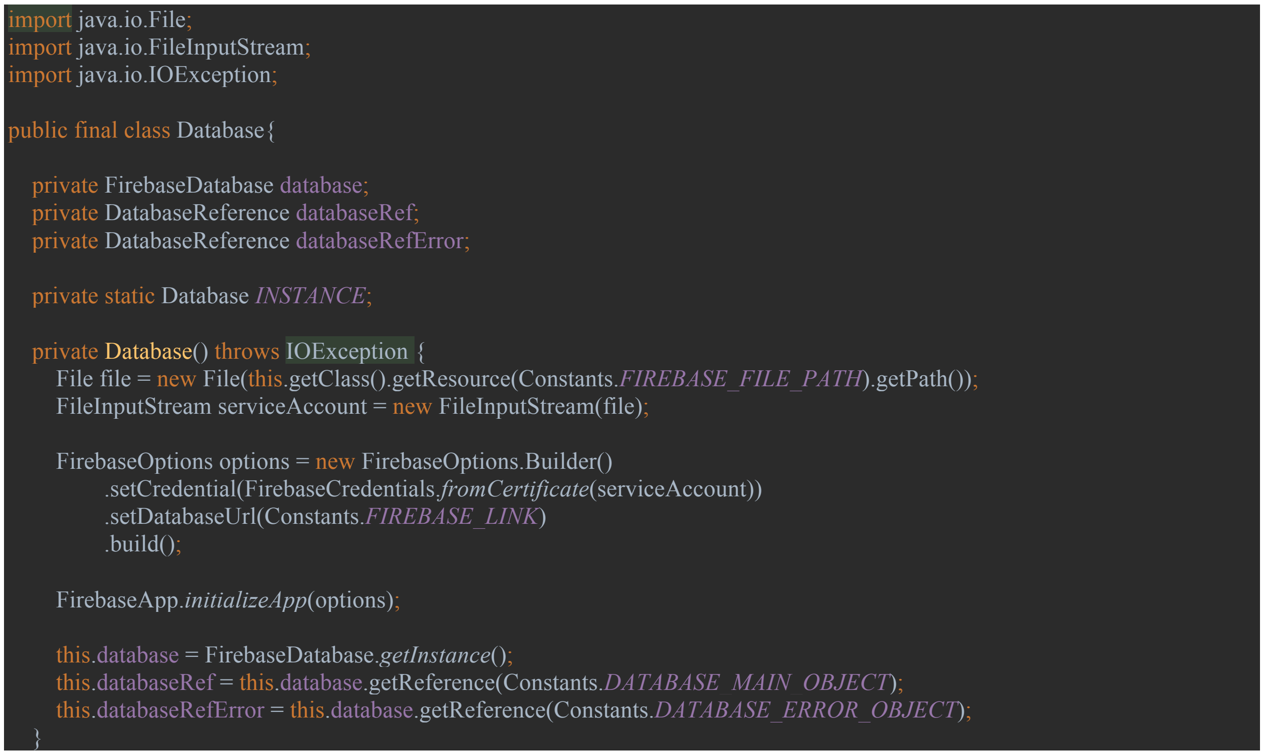
The ArrayList class extends the AbstractList and implements the List interface. Unlike standard Java arrays that are of a fixed length, ArrayList supports dynamic arrays that can grow and are created with an initial size. When this size is exceeded, the collection is automatically enlarged. The array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



```java
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

File 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. In this project, File I/O was not utilized directly. However, the real-time database library that was implemented used the File input library to read the certificate file to identify credentials in order to access the database (just like username and password, except in this case it has been hashed and stored in a JSON file).



```java
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;

public final class Database{

  private FirebaseDatabase database;
  private DatabaseReference databaseRef;
  private DatabaseReference databaseRefError;

  private static Database INSTANCE;

  private Database() throws IOException {
    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);
  }
```

Often an Exception is referred to a problem that arises during the execution of a program. Some of these exceptions are caused by user error (invalid input), others by programmer error (file not found), and others by physical resources that have failed in some manner (runs out of memory). Therefore, rather than disrupting the flow of the program or application by termination, exceptions can handle those cases. In this project, custom and premade (by JAVA library) Exceptions were used to handle the cases where a certain code might result in an error. For example, in the code above IOException was used in the Database constructor because there is a case where an error might arise from reading from a File. Another error that might arise is the file not existing at all, which if left unhandled might crash the application. Another example of custom RuntimeException is DirectionException which is used whenever the user input an invalid KeyCode.

```
public class DirectionException extends RuntimeException {

  public DirectionException() {
    super("Invalid Direction KeyCode.");
  }

  public DirectionException(String message) {
    super(message);
  }
}
```

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.

```
public final class GameController extends JPanel implements ActionListener, Runnable {

  private GameView gameView;
```

```
   private GameStatusBar gameStatusBar;
   private GameModel gameModel;

   private Timer timer;


   /**
    * Overridden method from Runnable interface to run the game on different Thread.
    */
   @Override
   public void run() {
      resetTimer();
   }
...
```

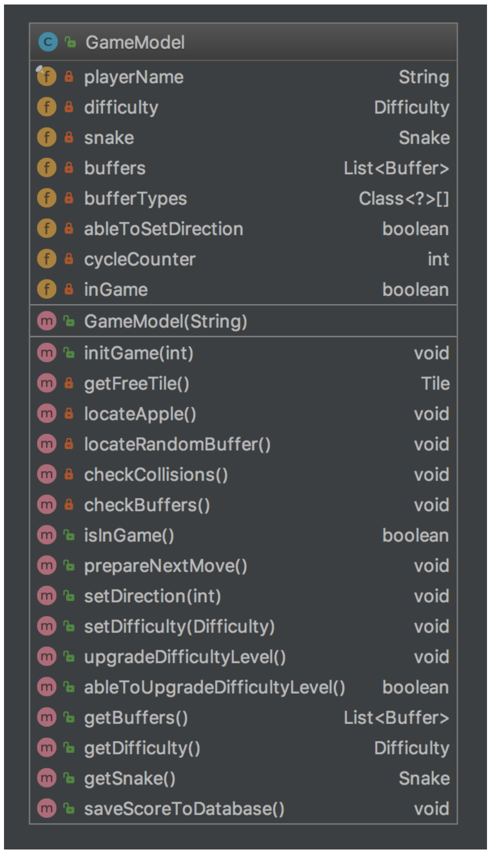
Junit:

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. As an example in the Snake class, the compareTo() method compared the scores of the current player with another and output one of three possible values, -1, 0, 1 for less than, equals, and greater than respectively. These cases were tested separately in the SnakeTest file to ensure that the proper value was obtained.

```
public class ScoreTest {
    …

    @Test
    public void compareToZero() throws Exception {
        Score model1 = new Score("Ammiel", 3);
        Score model2 = new Score("Jafar", 3);
        int expected = 0;
        int actual = model1.compareTo(model2);
        assertEquals(expected, actual);
    }

    @Test
    public void compareToOne() throws Exception {
        Score model1 = new Score("Ammiel", 4);
        Score model2 = new Score("Jafar", 3);
        int expected = 1;
        int actual = model1.compareTo(model2);
        assertEquals(expected, actual);
    }

    @Test
    public void compareToNegOne() throws Exception {
        Score model1 = new Score("Ammiel", 2);
        Score model2 = new Score("Jafar", 3);
        int expected = -1;
        int actual = model1.compareTo(model2);
        assertEquals(expected, actual);
    }
}
```

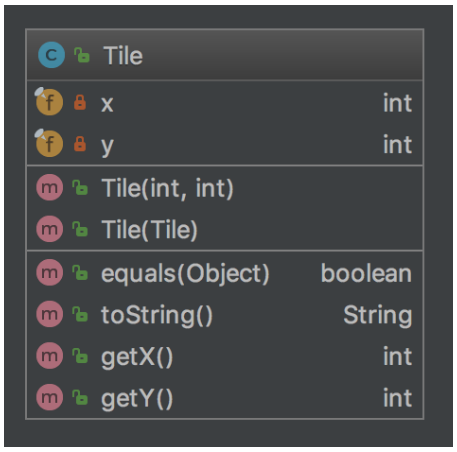
Class Diagrams of the Models, Exceptions, and Enums: (Note: Controller diagrams were omitted in this report since they were too large to add in.)

GameModel



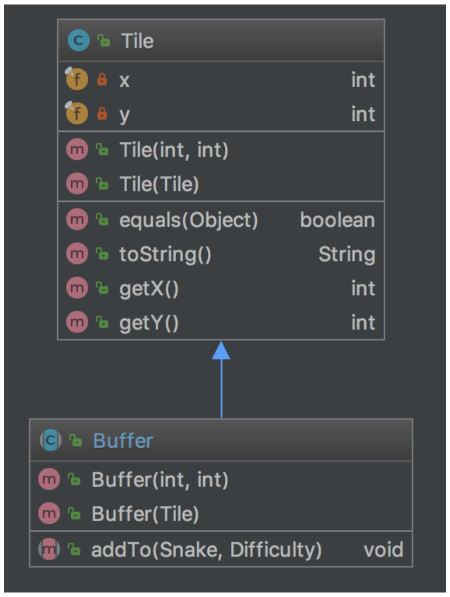
GameModel	
playerName	String
difficulty	Difficulty
snake	Snake
buffers	List<Buffer>
bufferTypes	Class<?>[]
ableToSetDirection	boolean
cycleCounter	int
inGame	boolean
GameModel(String)	
initGame(int)	void
getFreeTile()	Tile
locateApple()	void
locateRandomBuffer()	void
checkCollisions()	void
checkBuffers()	void
isInGame()	boolean
prepareNextMove()	void
setDirection(int)	void
setDifficulty(Difficulty)	void
upgradeDifficultyLevel()	void
ableToUpgradeDifficultyLevel()	boolean
getBuffers()	List<Buffer>
getDifficulty()	Difficulty
getSnake()	Snake
saveScoreToDatabase()	void

Tile



Tile	
x	int
y	int
Tile(int, int)	
Tile(Tile)	
equals(Object)	boolean
toString()	String
getX()	int
getY()	int

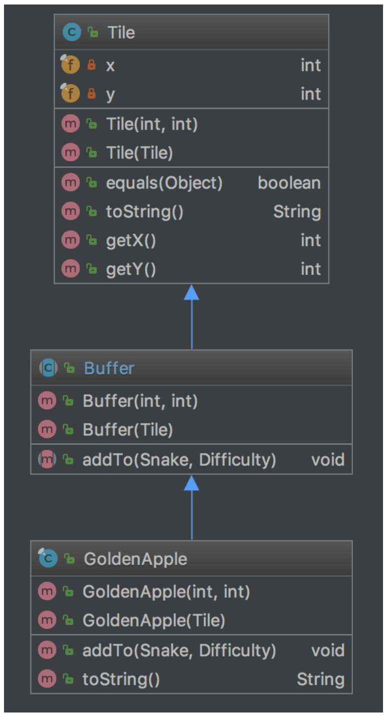
Buffer



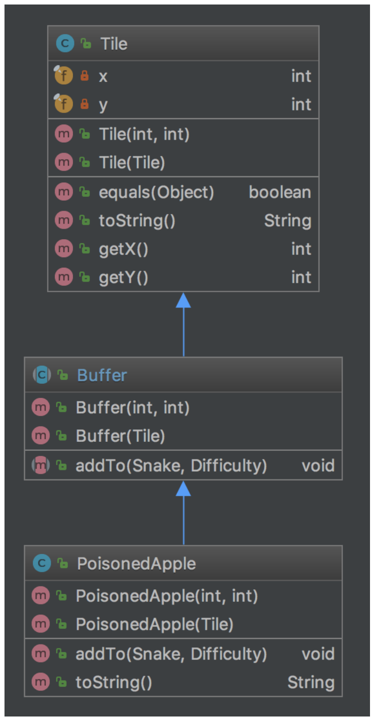
Tile	
x	int
y	int
Tile(int, int)	
Tile(Tile)	
equals(Object)	boolean
toString()	String
getX()	int
getY()	int

Buffer	
Buffer(int, int)	
Buffer(Tile)	
addTo(Snake, Difficulty)	void

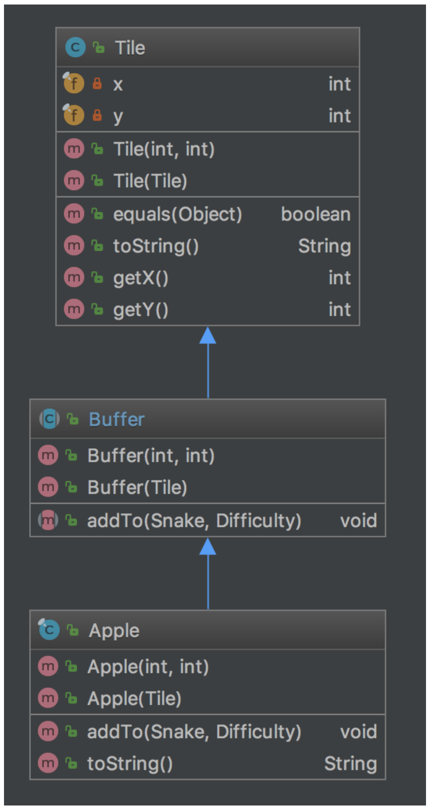
GoldenApple



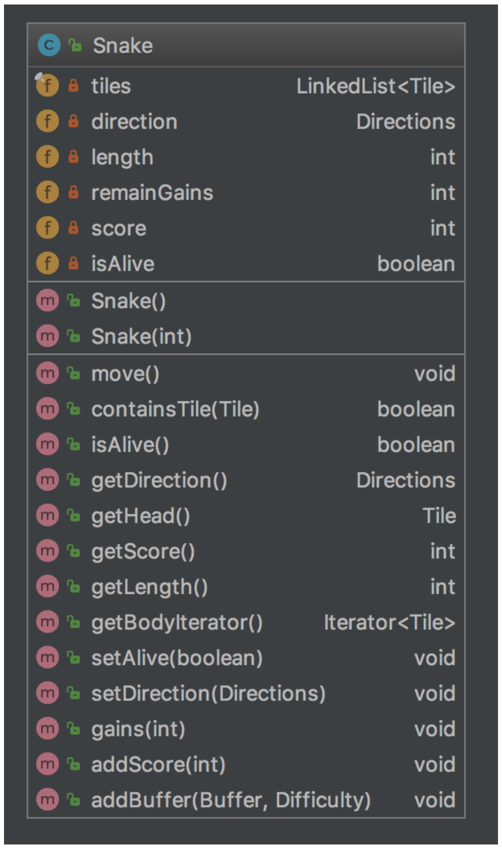
PoisonedApple



Apple

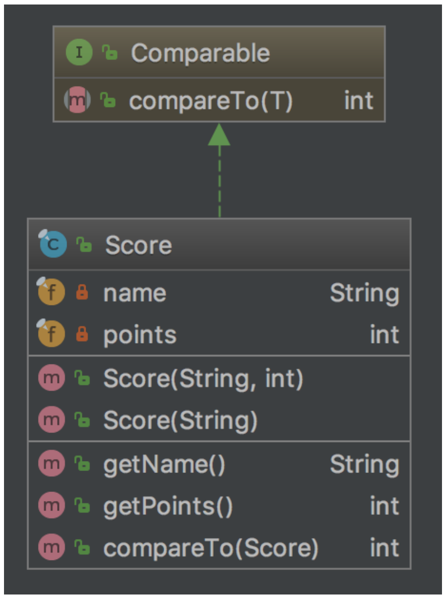


Snake



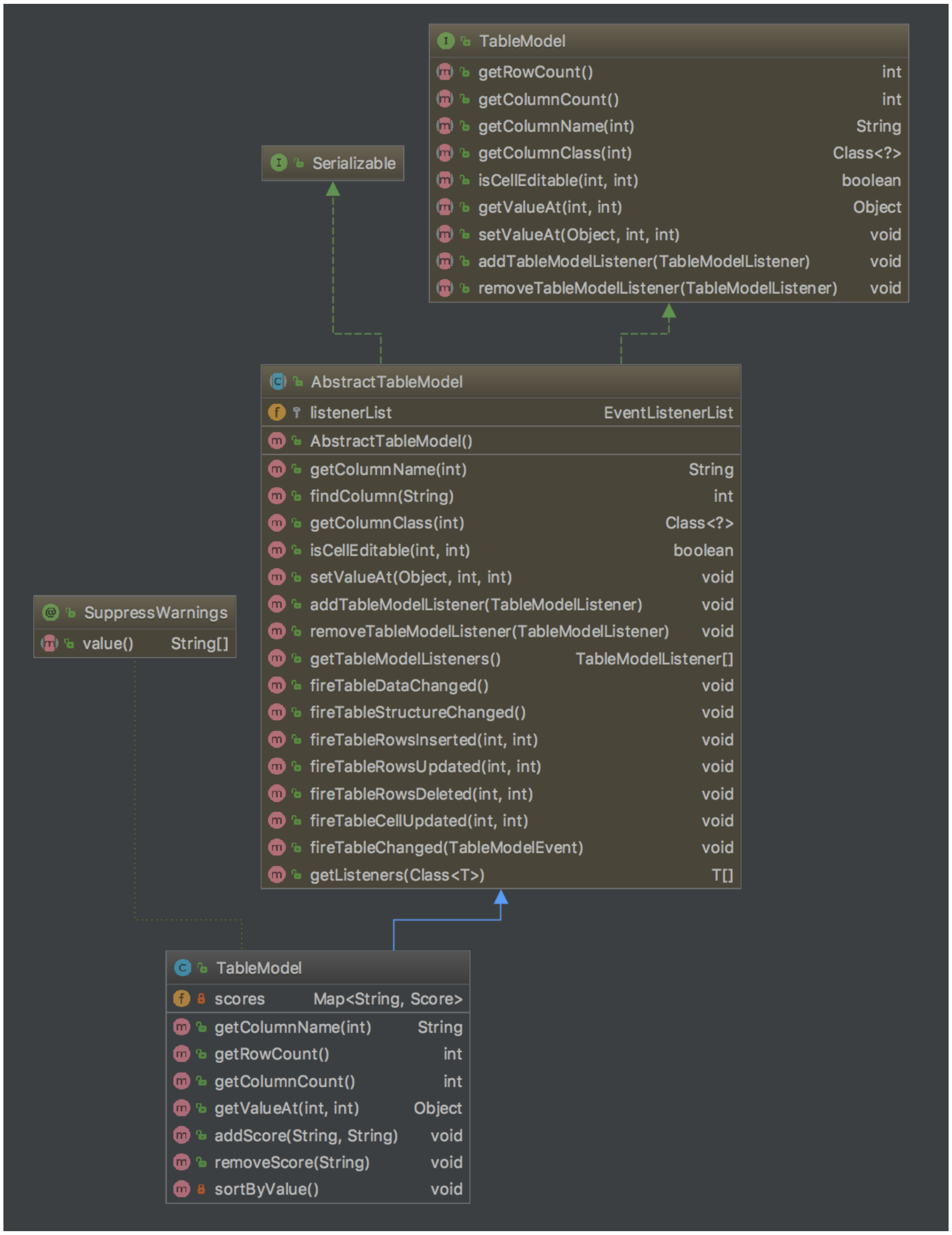
```
Snake
tiles                          LinkedList<Tile>
direction                      Directions
length                         int
remainGains                    int
score                          int
isAlive                        boolean
Snake()
Snake(int)
move()                         void
containsTile(Tile)             boolean
isAlive()                      boolean
getDirection()                 Directions
getHead()                      Tile
getScore()                     int
getLength()                    int
getBodyIterator()              Iterator<Tile>
setAlive(boolean)              void
setDirection(Directions)       void
gains(int)                     void
addScore(int)                  void
addBuffer(Buffer, Difficulty)  void
```

Score

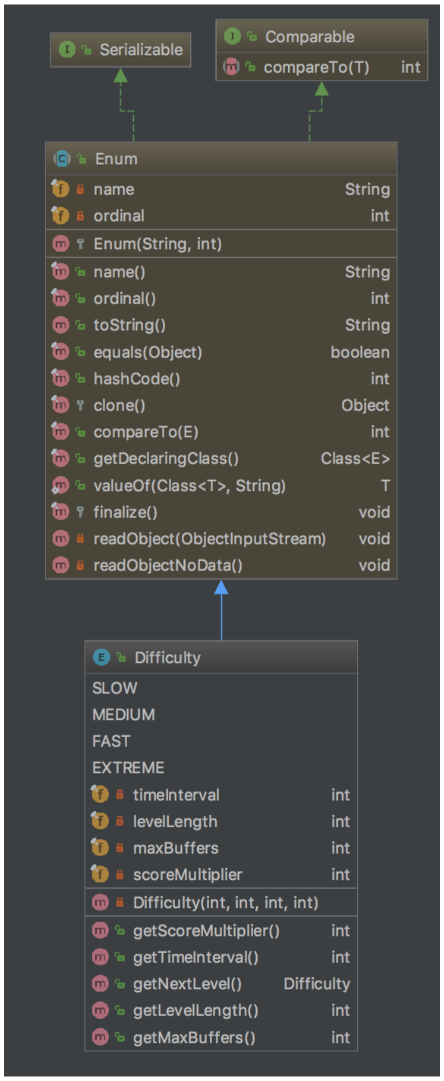


```
Comparable
compareTo(T)          int

Score
name                  String
points                int
Score(String, int)
Score(String)
getName()             String
getPoints()           int
compareTo(Score)      int
```

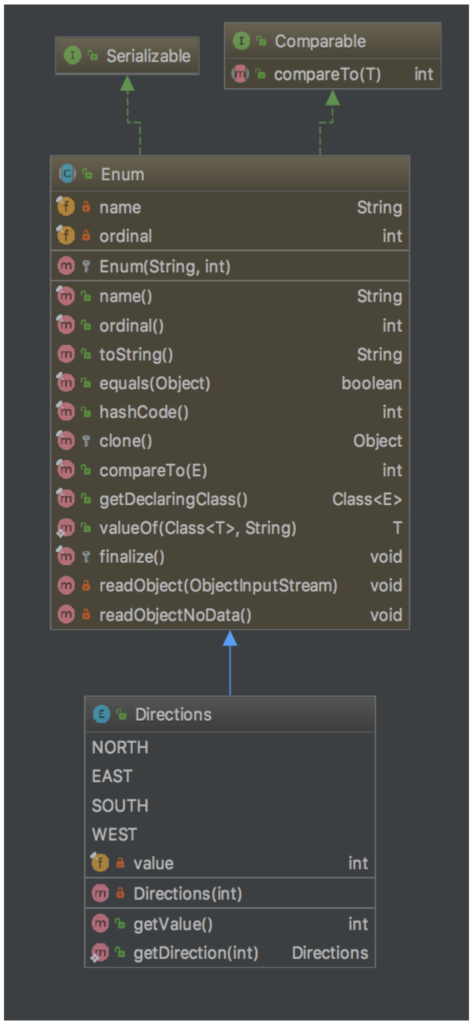
TableModel


TableModel
getRowCount() int
getColumnCount() int
getColumnName(int) String
getColumnClass(int) Class<?>
isCellEditable(int, int) boolean
getValueAt(int, int) Object
setValueAt(Object, int, int) void
addTableModelListener(TableModelListener) void
removeTableModelListener(TableModelListener) void
Serializable
AbstractTableModel
listenerList EventListenerList
AbstractTableModel()
getColumnName(int) String
findColumn(String) int
getColumnClass(int) Class<?>
isCellEditable(int, int) boolean
setValueAt(Object, int, int) void
addTableModelListener(TableModelListener) void
removeTableModelListener(TableModelListener) void
getTableModelListeners() TableModelListener[]
fireTableDataChanged() void
fireTableStructureChanged() void
fireTableRowsInserted(int, int) void
fireTableRowsUpdated(int, int) void
fireTableRowsDeleted(int, int) void
fireTableCellUpdated(int, int) void
fireTableChanged(TableModelEvent) void
getListeners(Class<T>) T[]
SuppressWarnings
value() String[]
TableModel
scores Map<String, Score>
getColumnName(int) String
getRowCount() int
getColumnCount() int
getValueAt(int, int) Object
addScore(String, String) void
removeScore(String) void
sortByValue() void

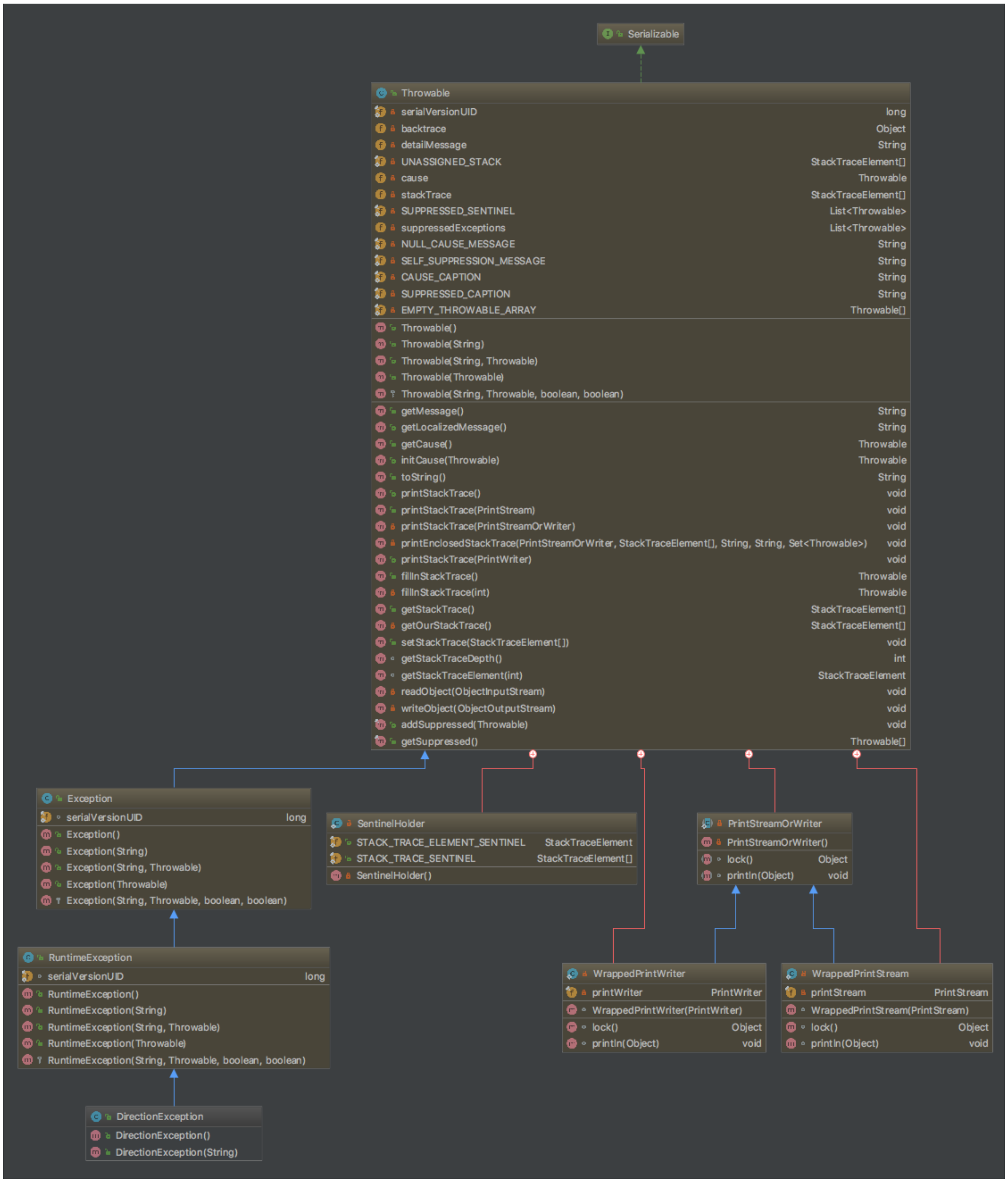
Difficulty



Directions



Exceptions


Serializable
Throwable
serialVersionUID long
backtrace Object
detailMessage String
UNASSIGNED_STACK StackTraceElement[]
cause Throwable
stackTrace StackTraceElement[]
SUPPRESSED_SENTINEL List<Throwable>
suppressedExceptions List<Throwable>
NULL_CAUSE_MESSAGE String
SELF_SUPPRESSION_MESSAGE String
CAUSE_CAPTION String
SUPPRESSED_CAPTION String
EMPTY_THROWABLE_ARRAY Throwable[]
Throwable()
Throwable(String)
Throwable(String, Throwable)
Throwable(Throwable)
Throwable(String, Throwable, boolean, boolean)
getMessage() String
getLocalizedMessage() String
getCause() Throwable
initCause(Throwable) Throwable
toString() String
printStackTrace() void
printStackTrace(PrintStream) void
printStackTrace(PrintStreamOrWriter) void
printEnclosedStackTrace(PrintStreamOrWriter, StackTraceElement[], String, String, Set<Throwable>) void
printStackTrace(PrintWriter) void
fillInStackTrace() Throwable
fillInStackTrace(int) Throwable
getStackTrace() StackTraceElement[]
getOurStackTrace() StackTraceElement[]
setStackTrace(StackTraceElement[]) void
getStackTraceDepth() int
getStackTraceElement(int) StackTraceElement
readObject(ObjectInputStream) void
writeObject(ObjectOutputStream) void
addSuppressed(Throwable) void
getSuppressed() Throwable[]
Exception
serialVersionUID long
Exception()
Exception(String)
Exception(String, Throwable)
Exception(Throwable)
Exception(String, Throwable, boolean, boolean)
SentinelHolder
STACK_TRACE_ELEMENT_SENTINEL StackTraceElement
STACK_TRACE_SENTINEL StackTraceElement[]
SentinelHolder()
PrintStreamOrWriter
PrintStreamOrWriter()
lock() Object
println(Object) void
RuntimeException
serialVersionUID long
RuntimeException()
RuntimeException(String)
RuntimeException(String, Throwable)
RuntimeException(Throwable)
RuntimeException(String, Throwable, boolean, boolean)
WrappedPrintWriter
printWriter PrintWriter
WrappedPrintWriter(PrintWriter)
lock() Object
println(Object) void
WrappedPrintStream
printStream PrintStream
WrappedPrintStream(PrintStream)
lock() Object
println(Object) void
DirectionException
DirectionException()
DirectionException(String)