UNIVERSITÀ DEGLI STUDI DI MESSINA

Corso di Laurea Triennale di Informatica - AA 2024-2025

Dipartimento di Scienze matematiche e informatiche, scienze fisiche e scienze della terra

PRESENTATO A - PROF. SALVATORE DISTEFANO

OBJECT ORIENTED PROGRAMMING

REPORT

Ansh Pandey
555869

TABLE OF CONTENTS

INTRODUCTION

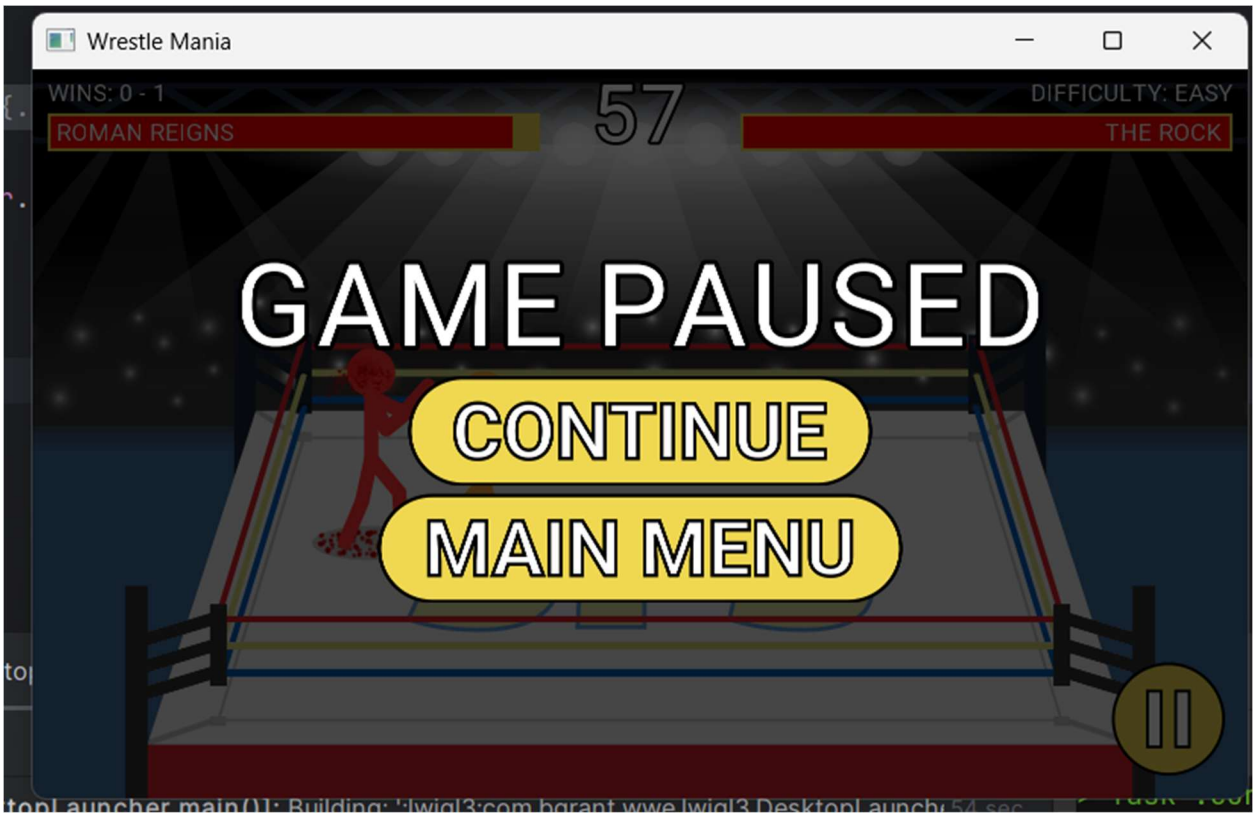
Wrestle Mania is a fun and exciting **fighting game** that brings the feel of professional wrestling right to your screen! This game is built using **object-oriented programming (OOP) principles**, **Java** using the **LibGDX** library. In the game, you can see how concepts such as **Reuse, Encapsulation, Abstraction, Information Hiding, Inheritance, Composition, Subtyping, Polymorphism, and Interfaces** are used. Additionally, features **like Exception Handling, File management, Threads, and Modularity** help make the game stable and easy to maintain.

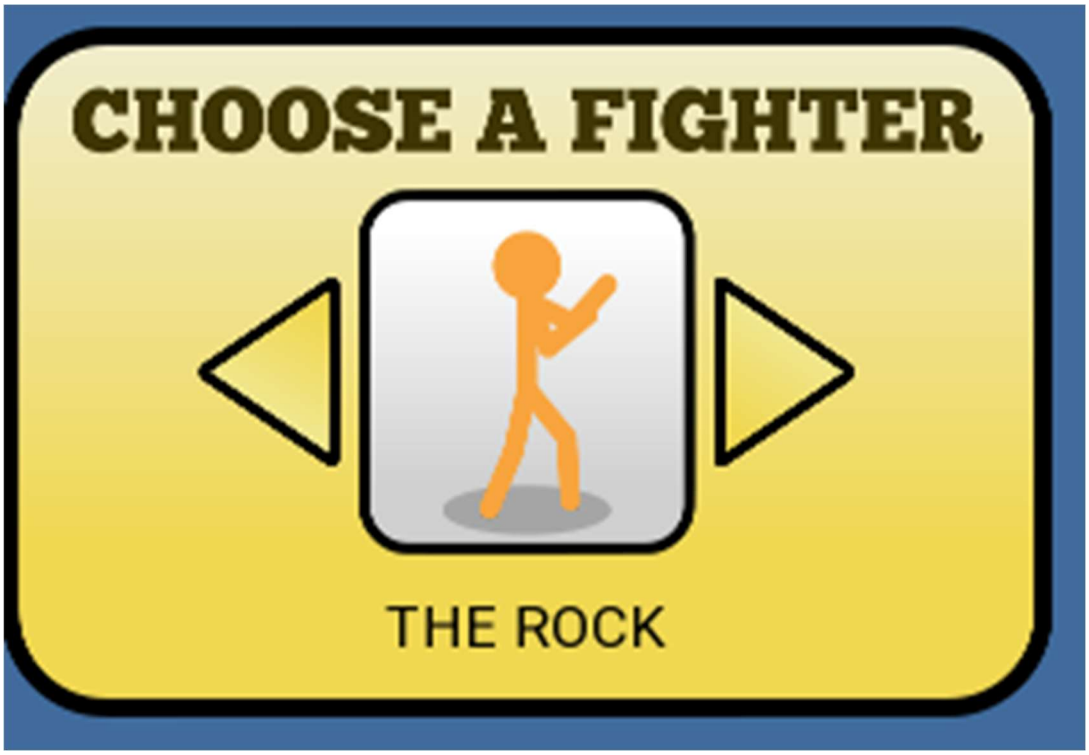
The game is designed to be very simple to play. It offers three **difficulty settings** and allows you to choose from **four** different **player options**. To keep things organized, the game has a **Loading Screen, Game Screen, Settings Screen, and Main Menu Screen**. One of the neat features made possible by LibGDX is the ability to switch to **full-screen mode** with just one click, along with several other cool, **easy-to-implement features**.

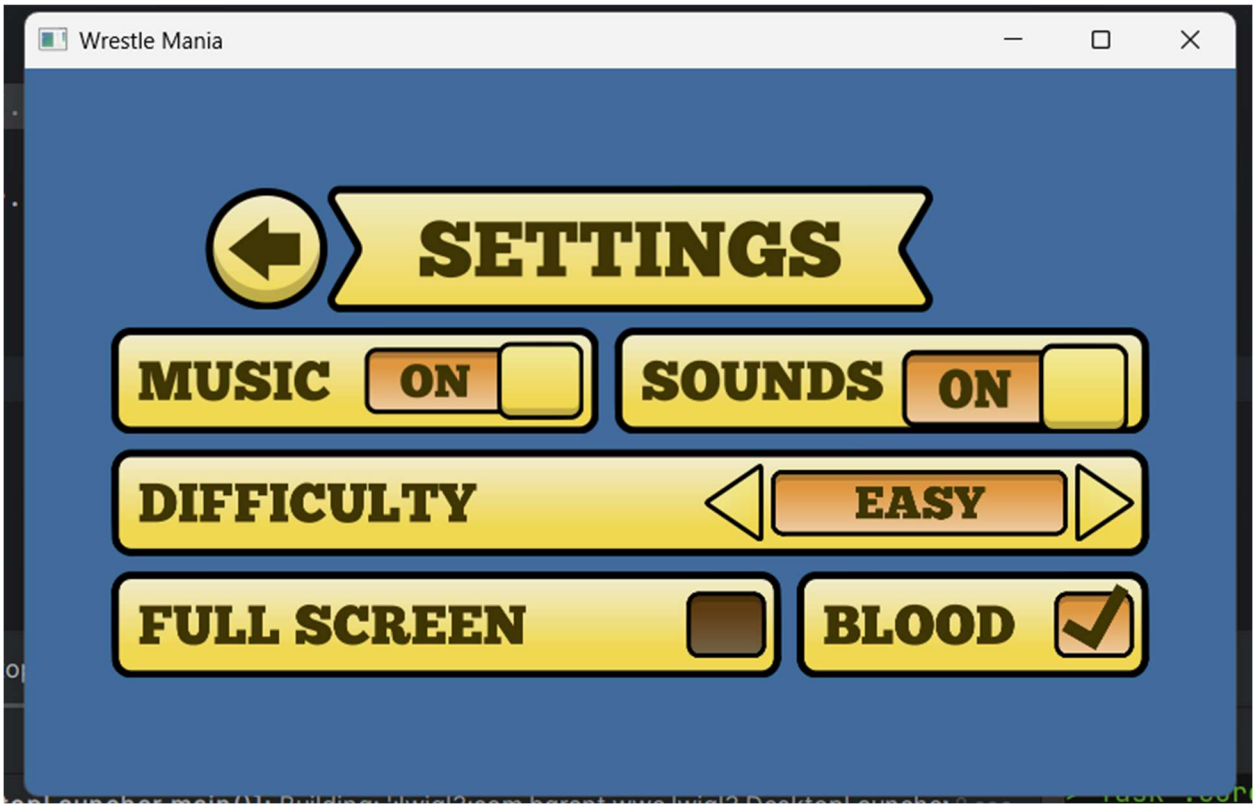
This **report** will explain the **design choices** and **technical details** behind **Wrestle Mania (WWE)**, showing how each **OOP concept** has helped shape a game that is not only **fun to play** but also well-**structured** and **efficient**.

GAME SCREENSHOTS


Wrestle Mania
WINS: 0 - 0
50
DIFFICULTY: EASY
ROMAN REIGNS
THE ROCK
SFS


Wrestle Mania
WINS: 0 - 1
57
DIFFICULTY: EASY
ROMAN REIGNS
THE ROCK
GAME PAUSED
CONTINUE
MAIN MENU


CHOOSE A FIGHTER
THE ROCK

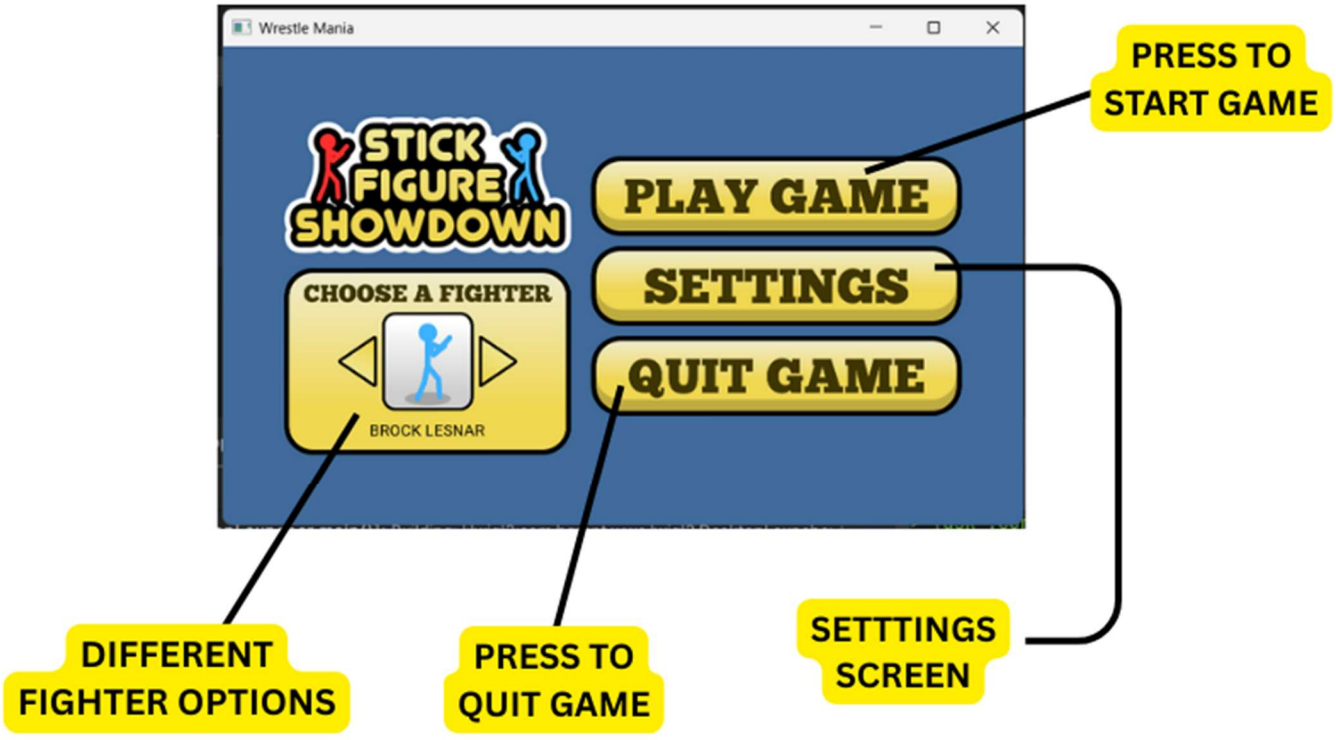

Wrestle Mania
SETTINGS
MUSIC
ON
SOUNDS
ON
DIFFICULTY
EASY
FULL SCREEN
BLOOD

Game Features Overview

This section highlights the key **features** of the **Wrestle Mania** game, which offers an engaging and immersive gameplay experience. Each feature has been carefully designed to make the **game** more **interactive** and exciting, while also ensuring it stays **intuitive** and **user-friendly**.

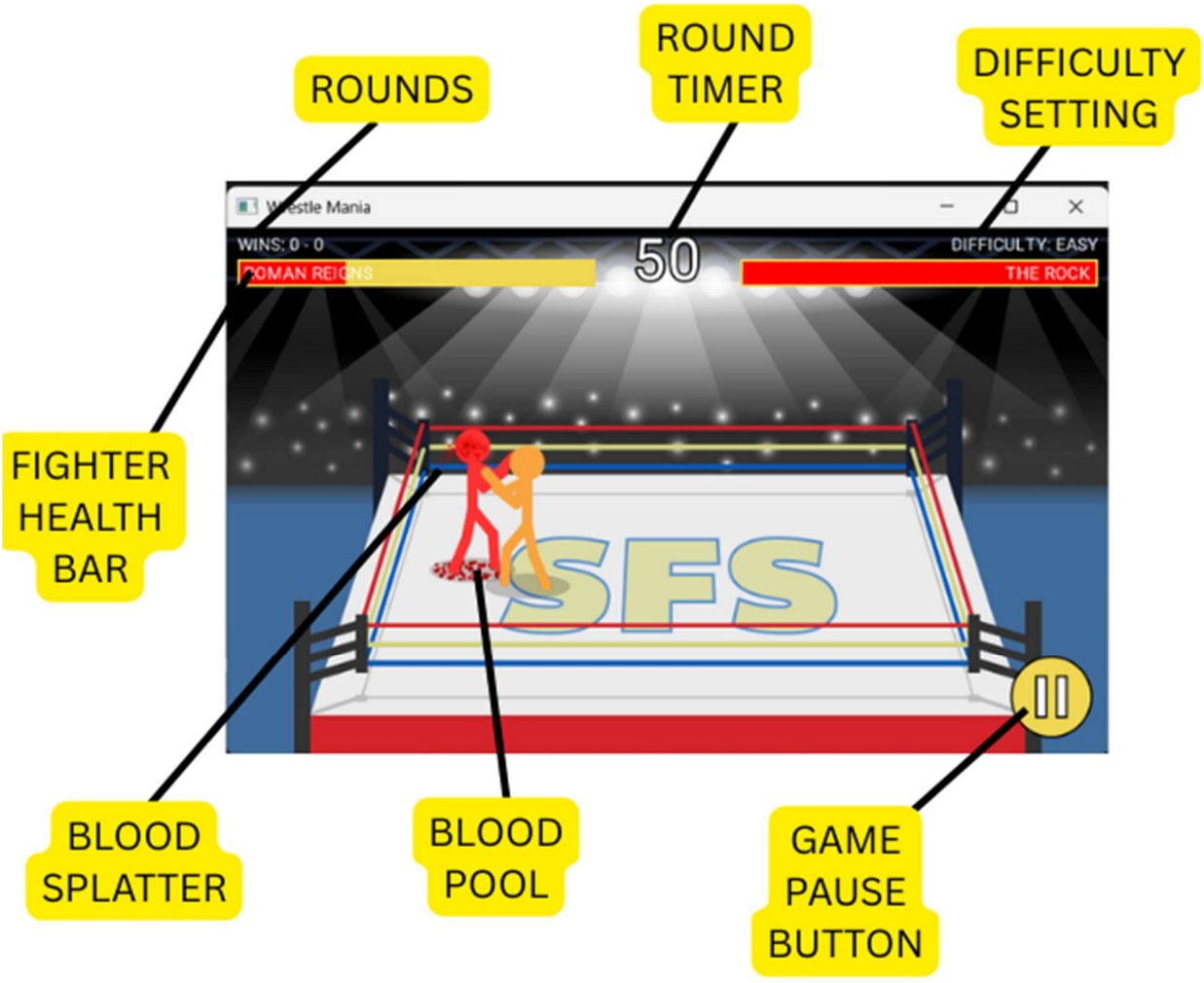
1. Main Menu Screen

Upon starting the game, players are presented with a **Main Menu** screen that allows them to navigate to various game modes and settings. The main menu offers the following options:



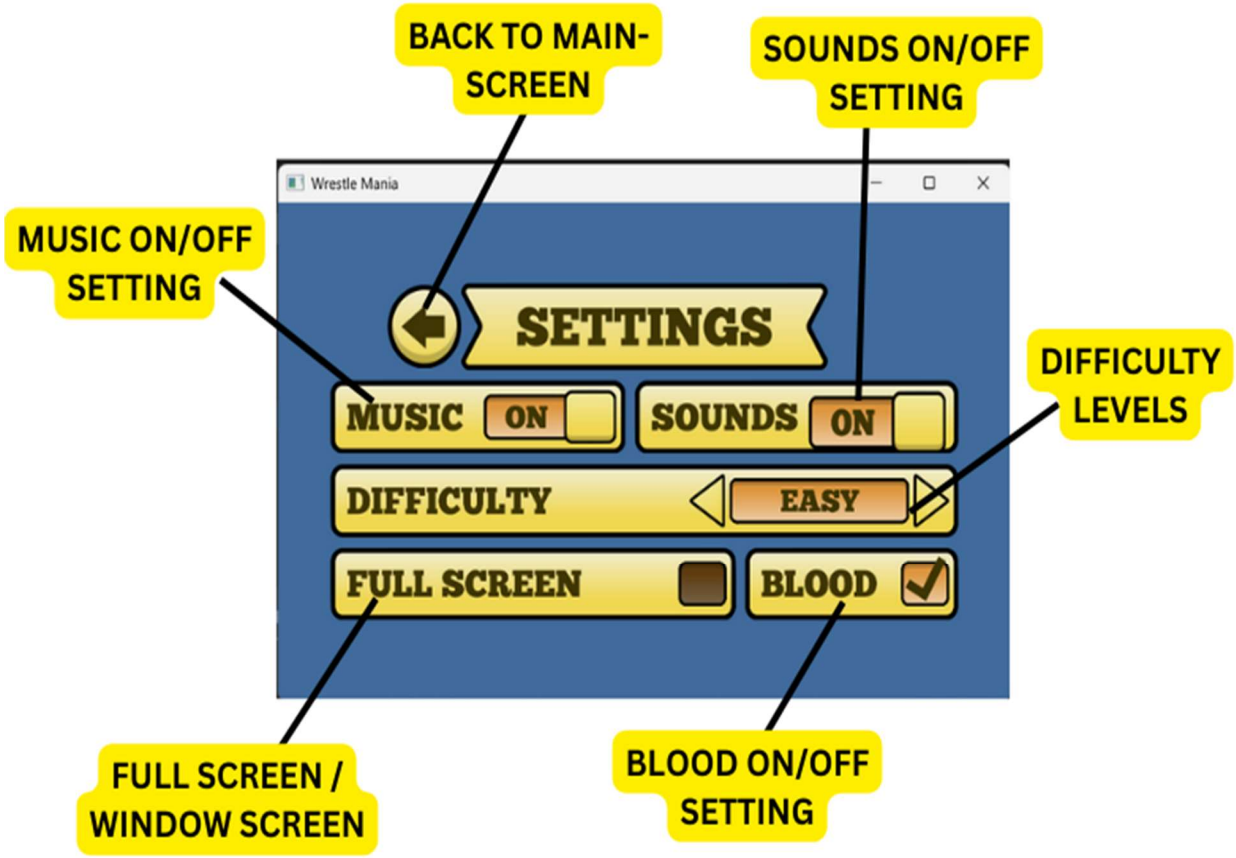
2. Game Screen

Once the player has selected their fighter and the match begins, they are taken to the **Game Screen**. This screen serves as the main interface during the battle, displaying important elements such as the **fighters**, their **health bars**, and the **environment**. The **Game Screen** allows players to actively engage in combat and utilize their **fighter's moves like Punch , Kick & Block** to defeat the opponent. Key features of the Game Screen include:



3. Settings Screen

The **Settings Screen** allows players to **customize their gameplay experience** to suit their preferences. Accessible from the **main menu**, this screen offers several adjustable options to enhance both the audio and visual elements of the game. The key features of the Settings Screen include:



OOP Concepts

Object-Oriented Programming (OOP) is a programming paradigm that organizes software around objects, which are instances of classes. It separates implementation details (such as data structures) from core properties (functions for accessing or manipulating the data), enhancing scalability and design for complex systems.

In OOP, an **abstract data type (ADT)** or object is defined by its values and the operations that can be performed on it. This is achieved through **encapsulation**, which hides the internal workings and exposes only necessary functionality to users.

Key Concepts of OOP:

- **Encapsulation**: Bundling data (attributes) and methods (functions) into an object. This protects data from unauthorized access and ensures controlled interaction with it.
- **Abstraction**: Hiding complex details and exposing only essential features, reducing system complexity for users.
- **Inheritance**: A new class inherits properties and behaviors (methods) from an existing class, promoting code reuse and creating hierarchical relationships between classes.
- **Polymorphism**: The ability to treat objects from different classes as objects of a common superclass, allowing methods to work with various object types and increasing code flexibility.

OOP Methodology: The OOP process involves these steps:

1. **Identify objects**: Determine relevant objects for the system at the desired abstraction level.
2. **Define object behavior**: Specify actions each object can perform.
3. **Establish relationships**: Group and define relationships between objects.
4. **Implement objects**: Develop objects iteratively, refining the design with each cycle.

This iterative process allows refinement of the system, without necessarily following a top-down approach.

Benefits of OOP: OOP offers several advantages over traditional procedural programming for developing complex systems:

- **Modularity**: OOP organizes code into classes, which act as modular components of the system.
- **Module Cohesion**: Each class represents a single entity, ensuring that data and behaviors are related.
- **Decoupling**: Objects are decoupled since methods operate on internal data, improving maintainability.
- **Information Hiding**: Data and implementation details are hidden from external code, promoting secure and stable interactions.
- **Reusability**: Inheritance allows new classes to reuse existing ones, minimizing code duplication.
- **Extensibility**: Polymorphism makes it easy to add new features or functionality without major changes to existing code.

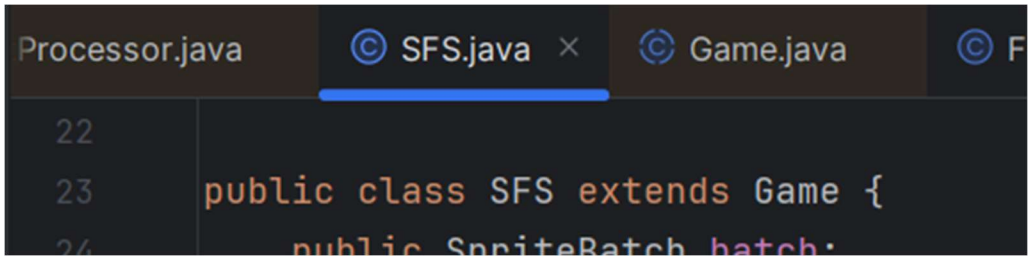
INHERITANCE

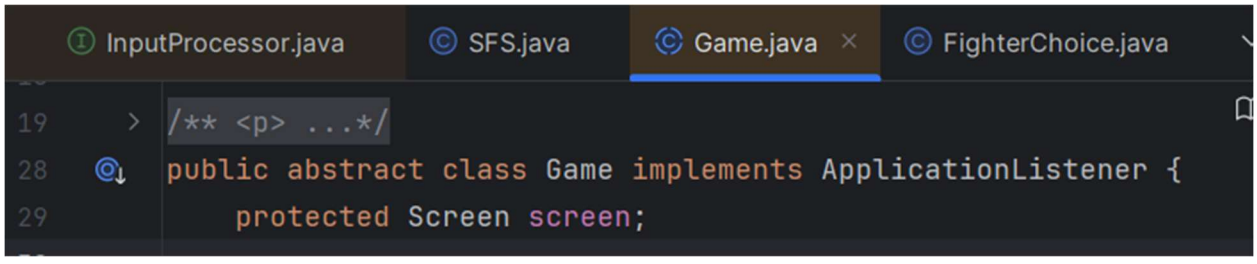
Inheritance is a fundamental concept of **Object-Oriented Programming** (OOP) where a **new class** is created by acquiring the members (fields and methods) of an existing class, and possibly extending them with new or modified functionalities. This principle allows **one class**—referred to as the ***subclass* or *child class***—to **inherit** the **properties** and **behaviors** of another class, known as the ***superclass* or *parent class***. In Java, inheritance is implemented using the **extends keyword for classes and the implements keyword for interfaces**.

By enabling code reuse, **inheritance** not only **saves development time** but also **promotes maintainable** and **modular software design**. Developers can build upon proven, well-tested code, **reducing redundancy and potential bugs**. This makes systems **more efficient** to implement and **easier to evolve over time**.

*****CODE SNIPPETS*****

Extending LibGDX Game Class



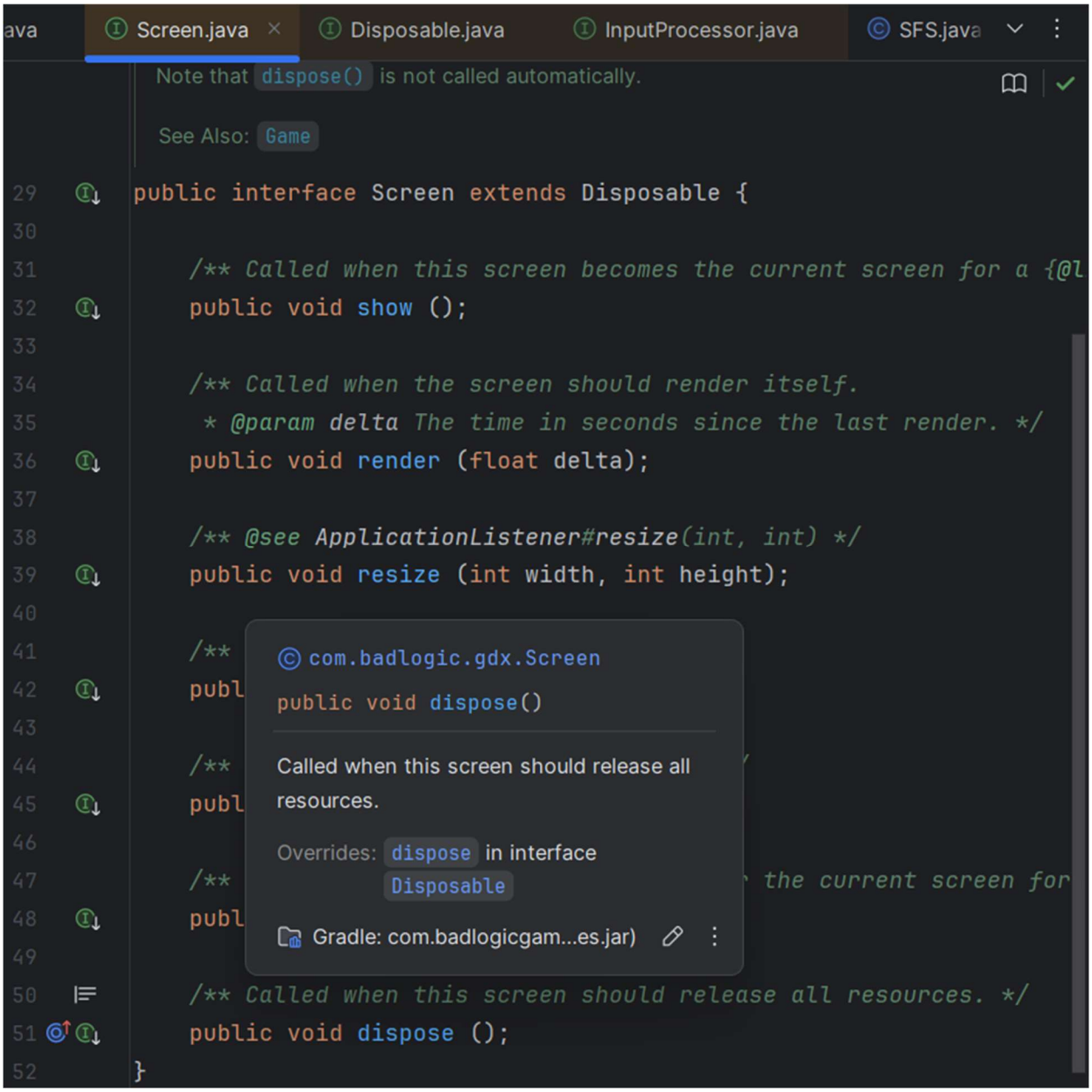


The **custom SFS** class inherits from **Game** class, which provides **built-in screen management**. The **overridden create()** method is called when the **application starts**, allowing **initialization of the first screen**. This use of **inheritance allows reuse** of LibGDX's screen switching system.

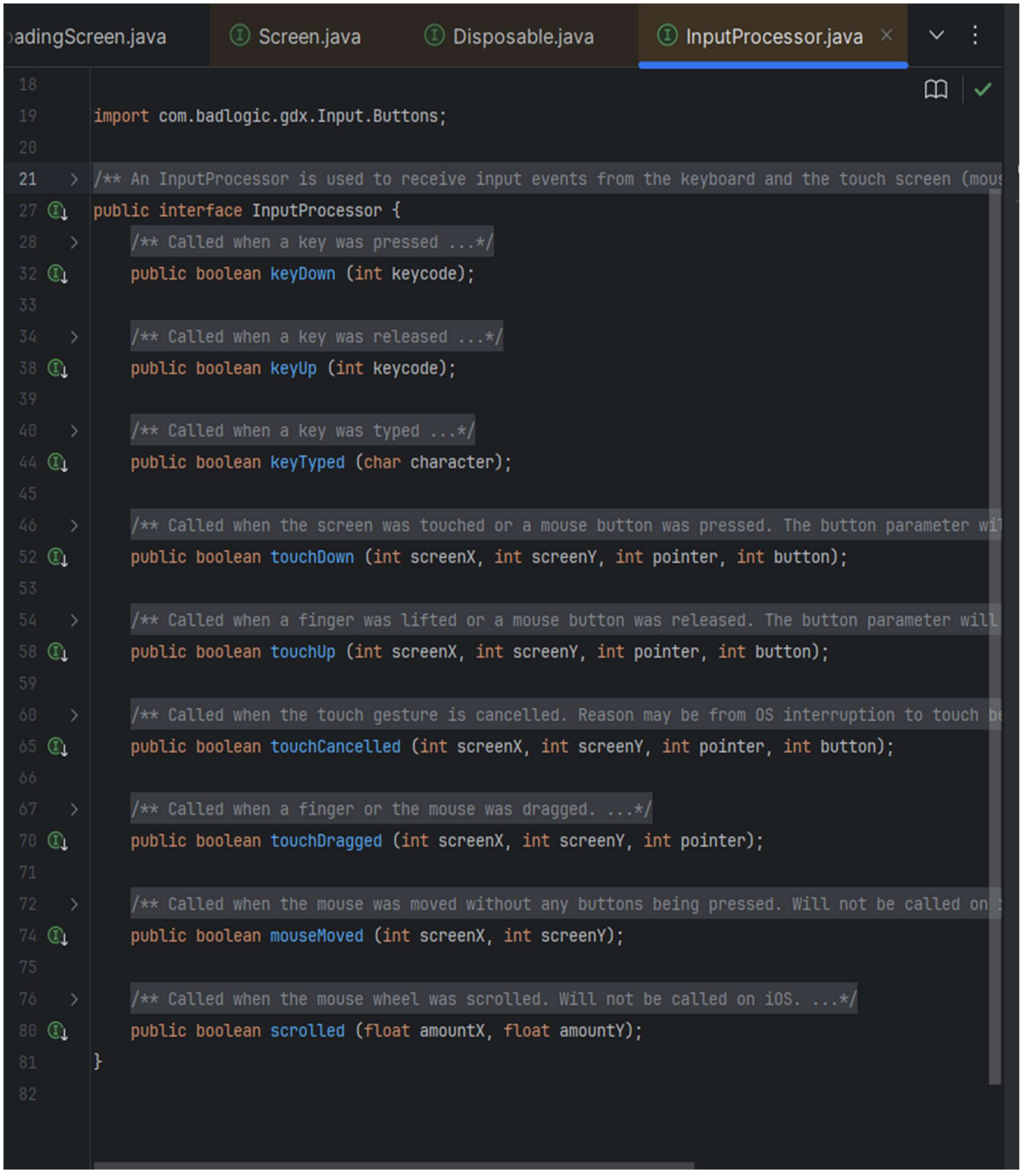
Interface-Based Inheritance in LibGDX

LibGDX uses interface-based inheritance to promote **modularity** and **flexibility**. Interfaces like **Screen** and **InputProcessor** allow classes to inherit a **set of method** contracts they must implement.

SCREEN INTERFACE

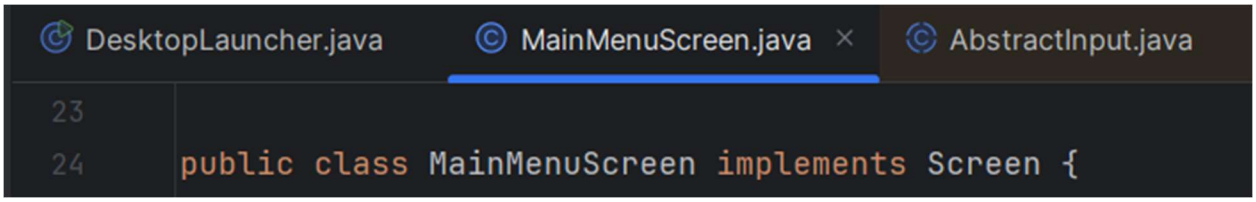


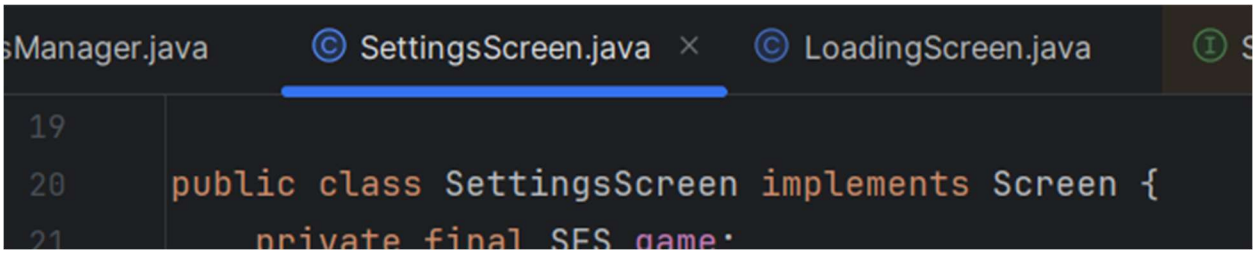
INPUT PROCESSOR INTERFACE



Screen Implementations

In the project, multiple screens such as **Main Menu, Settings, Loading, and Game screens** implement the **Screen interface**. Each class provides its own implementation of the required methods, enabling custom behavior.





Multiple Interface Inheritance in GameScreen

The **GameScreen** class demonstrates **multiple inheritance** by implementing both **Screen** and **InputProcessor**. This allows the class to **manage screen events and handle user input such as key presses**.

Benefits of Using Inheritance

- **Code Reuse:**

  -The SFS class reuses the Game class's screen management.

  -GameScreen, SettingsScreen, and other screen classes reuse the structure provided by the Screen interface.

  - GameScreen also reuses the input-handling methods specified in the InputProcessor interface.

- **Extensibility:** New characters or screens can be added by extending existing classes.
- **Consistency:** All screens follow a standard structure defined by the Screen interface.

Difference Between Extends and Implements

- **Extends** is used when a class inherits from another class. It gains access to all non-private fields and methods and can override them.

- **Implements** is used when a class agrees to follow a contract defined by an interface. The class must implement all methods declared in the interface.

Conclusion

Inheritance in the **Project** allows for powerful code reuse and a modular, structured approach to **game development**. By using both **class inheritance (extends) and interface implementation (implements),** we are able to create flexible, scalable, and maintainable code that forms the backbone of our **game architecture**.

ABSTRACTION

Abstraction is a cornerstone of **Object-Oriented Programming** (OOP) that lets us tame complexity by **exposing only** the essentials of a **component and hiding its inner workings**. In a large game like mine, abstraction helps in what each part does—**screens**, **game objects**, **resource** loaders—without getting bogged down in how each **detail** is implemented. This yields code that is **modular**, **reusable**, and **easier to maintain**.

In Java, we achieve abstraction primarily via:

- **Abstract classes** – classes that cannot be instantiated and may declare abstract methods (no body) that subclasses must implement.
- **Interfaces** – pure contracts defining method signatures; implementing classes supply the behavior.

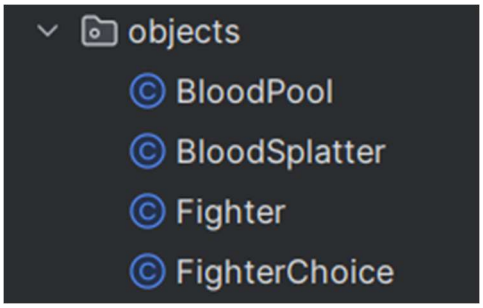
Benefits of Abstraction

Abstraction provides several key benefits in game development:

- **Simplifies Complex Problems**: By hiding unnecessary details, abstraction allows you to focus on high-level functionality, making complex tasks easier to manage.
- **Algorithm Independence**: The logic behind game mechanics can be analyzed independently of the language or technology used for implementation.
- **Flexibility in Development**: Using abstract data structures, game systems can be built without worrying about the specific algorithmic details, allowing for easier updates or optimisations.

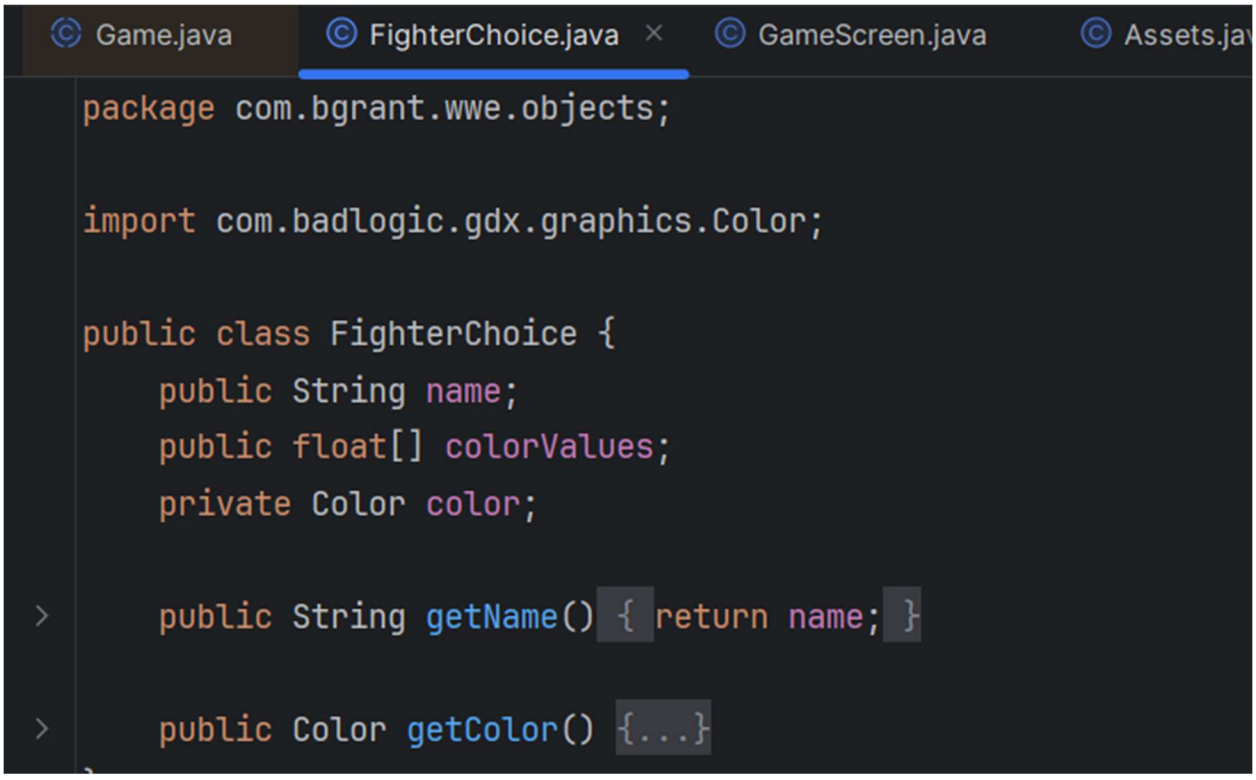
1. Object Abstraction

Abstraction is used to represent in-game entities and effects, providing a clear distinction between *what* an object is and *how* it behaves. The key classes are located in the **objects** package, each of which represents a specific **entity or effect in the game**.



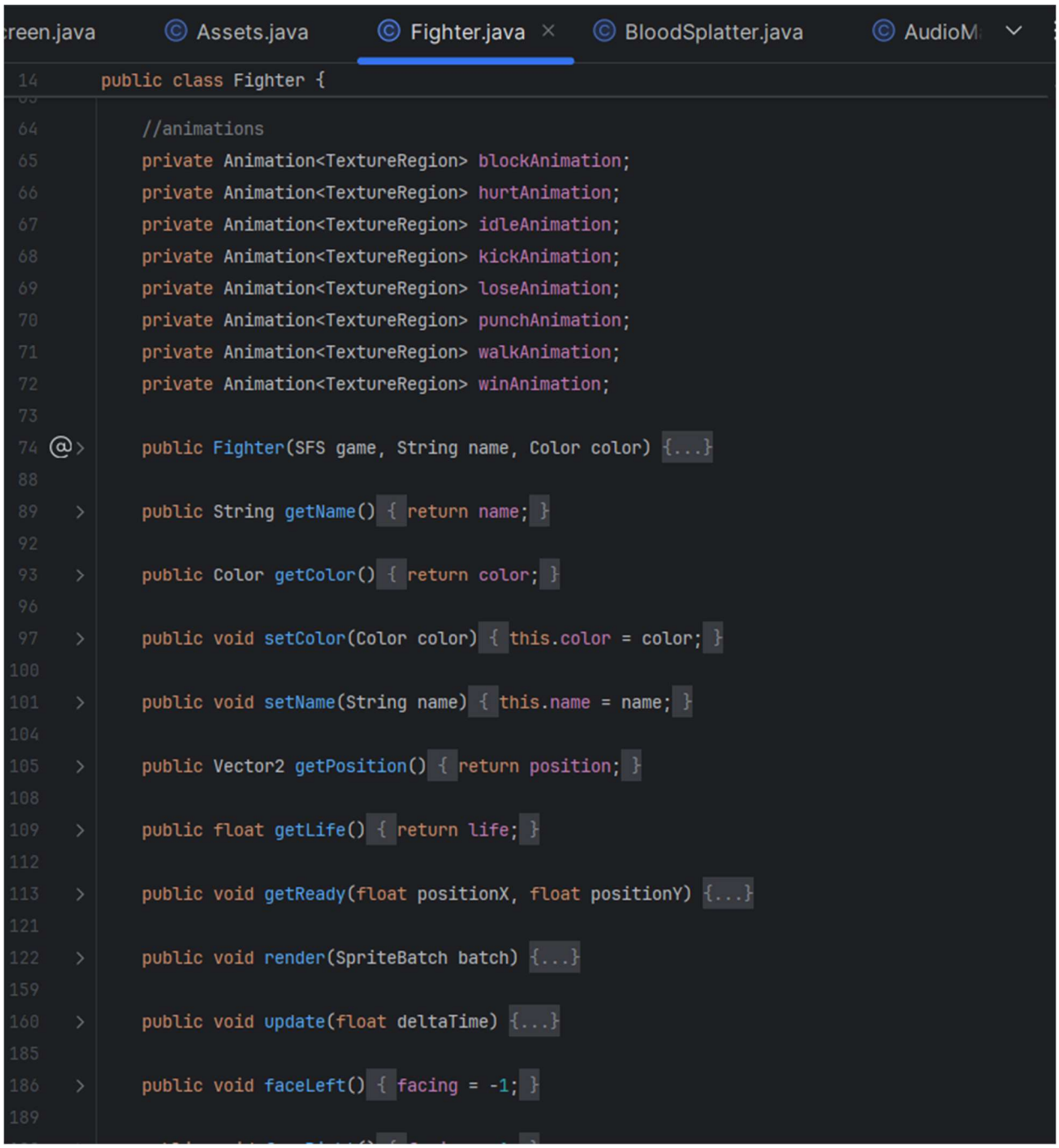
FighterChoice

The **FighterChoice** class abstracts the logic for creating different types of **fighters**. By using this **abstraction**, the game logic does not need to know the details of how fighters are created, just that it can select **between different fighter** types using a **JSON file** .



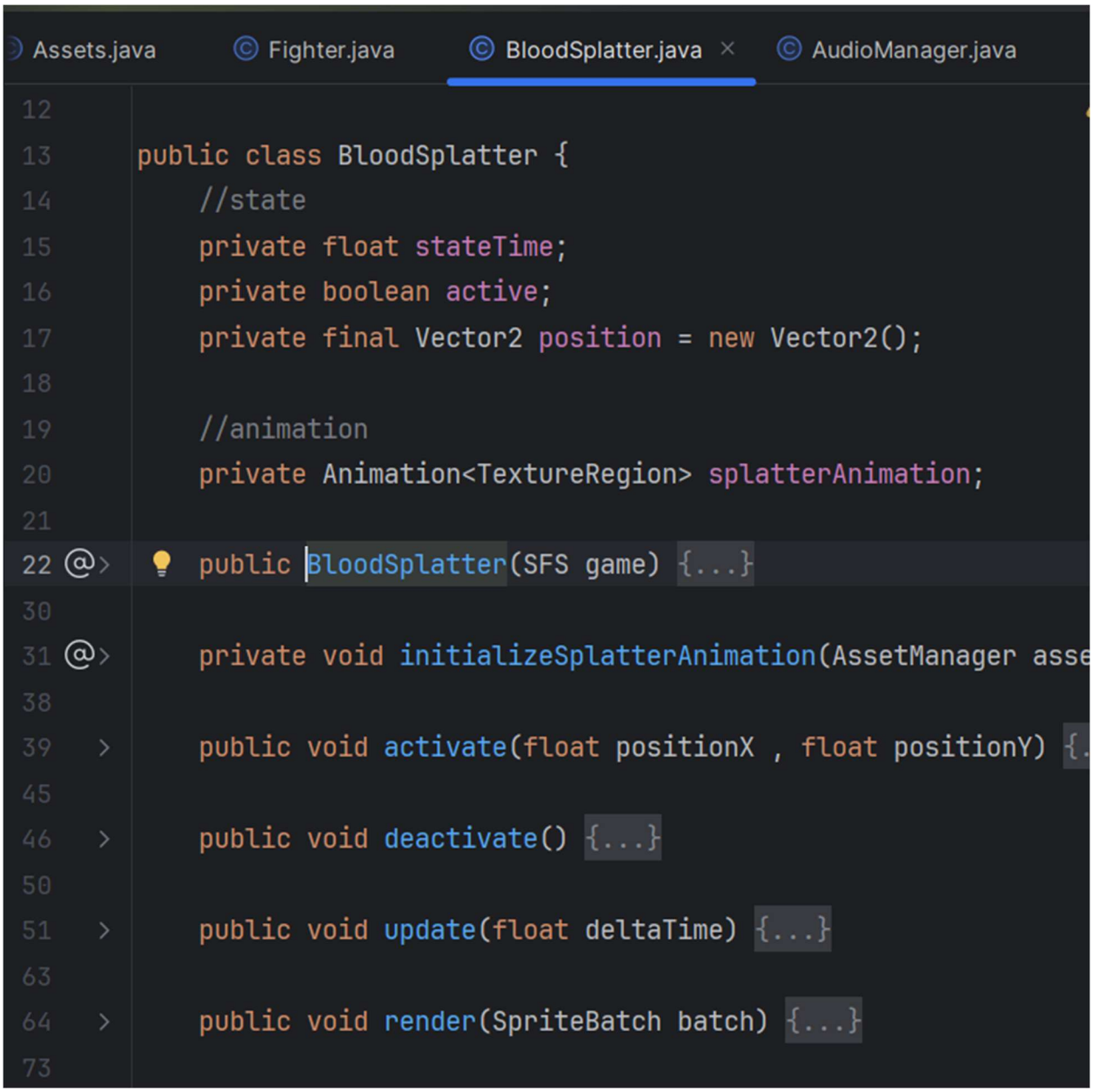
Fighter

The **Fighter** class serves as an **abstract** base class for all **combatants** in the game. It encapsulates the shared properties and behaviors of fighters, such as **health**, **attack , block , power**, and **actions** (like **attacking and taking damage**). The actual fighter types, like **PlayerFighter** and **EnemyFighter**, extend Fighter and implement specific attack behaviors.



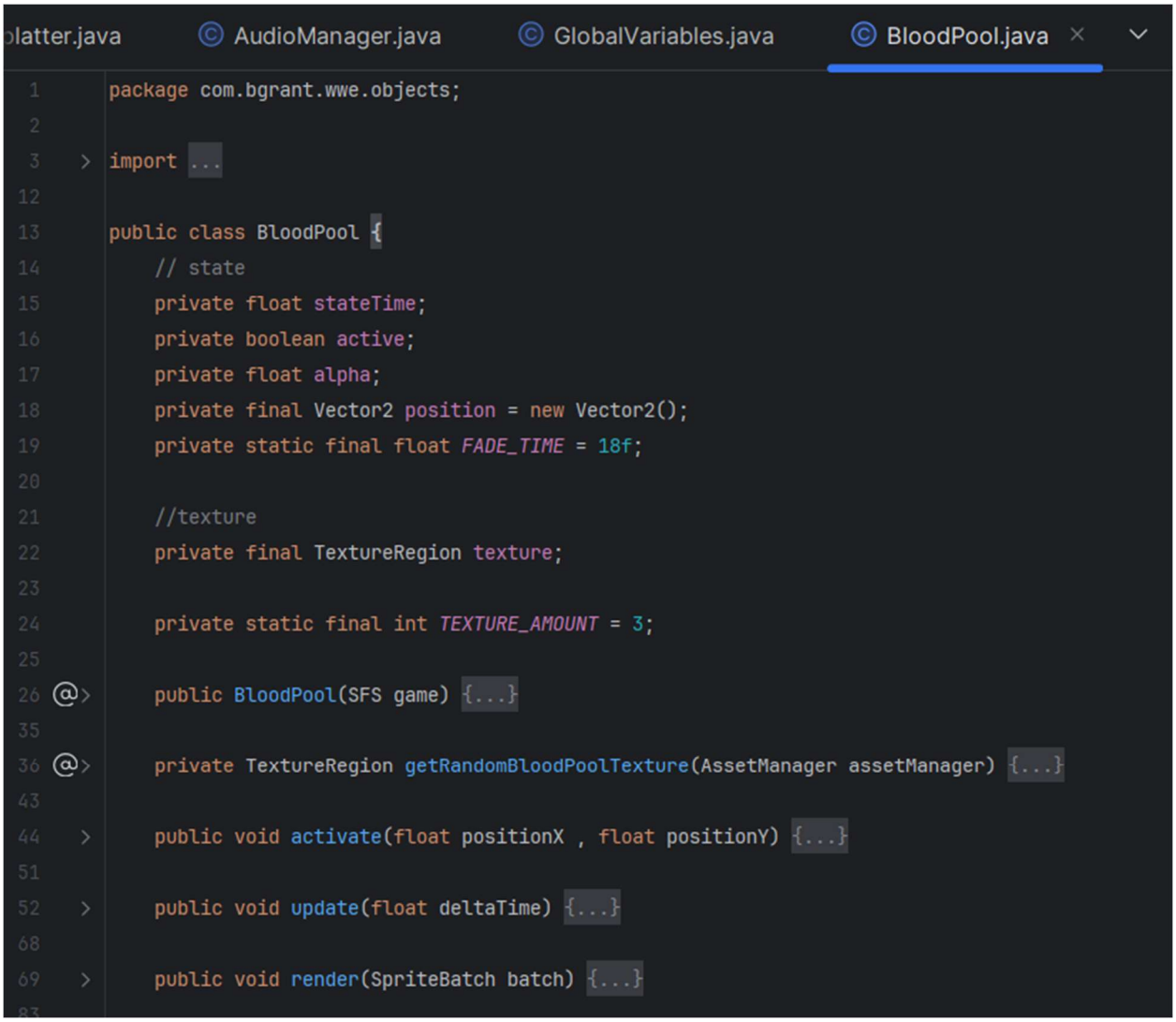
BloodSplatter

The **BloodSplatter** class is responsible for defining and managing the behavior of each **blood effect**. Each instance of **BloodSplatter represents a single blood splatter** that appears on the **screen when a fighter** takes **damage**. It controls the **animation, lifetime, and position** of the splatter.



BloodPool

The **BloodPool** class is responsible for managing the **blood that pools** on the **ground** after a fighter lands a significant hit. This pooling effect is not an **instant visual** but rather a **blood stain** that stays for a **few seconds**, slowly **fading away over time**. The pool of blood stays in **place on the ground** for around **18** seconds, gradually **becoming less visible** as time passes.

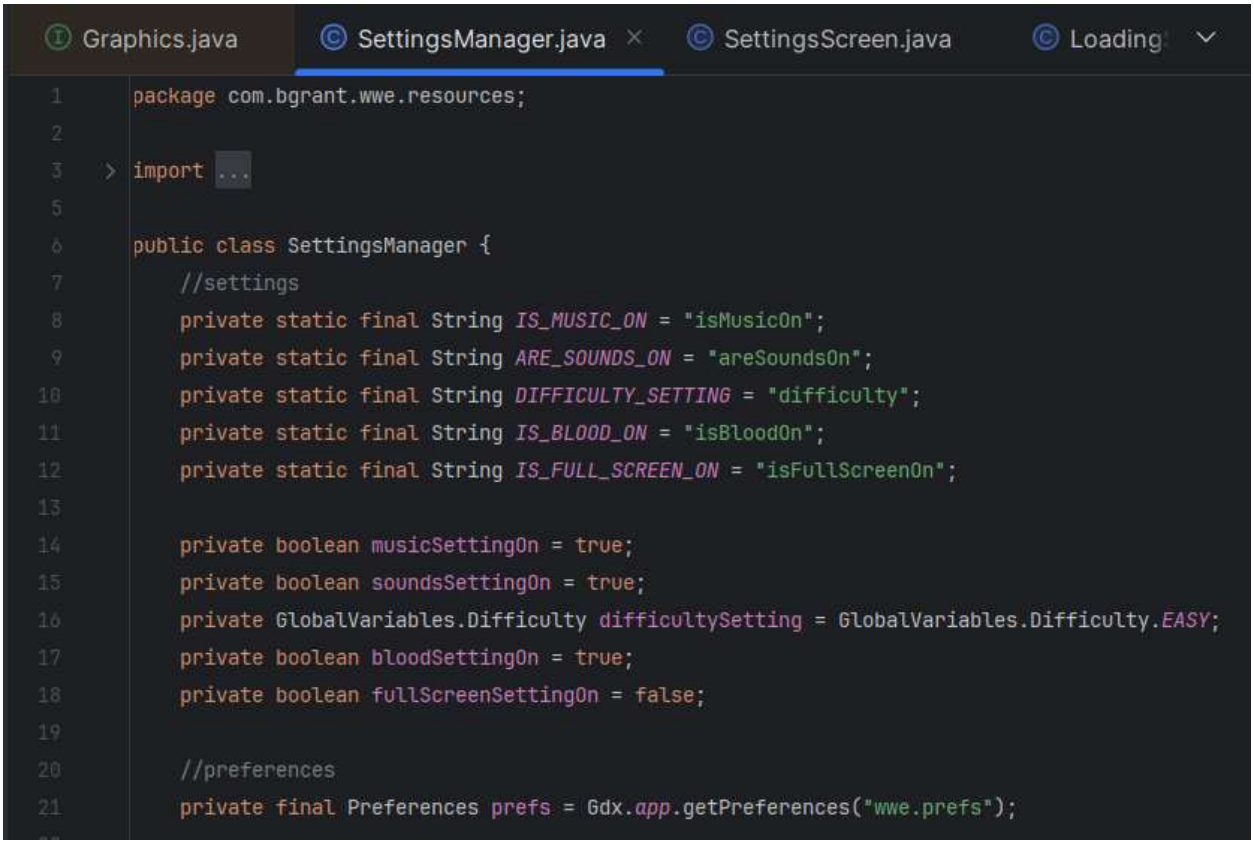


2.Resource Abstraction

The **management** of game resources, such as **images**, **sounds**, and **settings**, can become **complex** and difficult to maintain. To **simplify** this, **abstraction** is used in the **resources package** to **manage assets, audio playback, and configuration settings**.

SettingsManager

The **SettingsManager** class handles the **configuration** of game settings (such as volume , sounds , full-screen , blood visuals). It uses **LibGDX's** Preferences **API** to save and **load settings**.



Assets

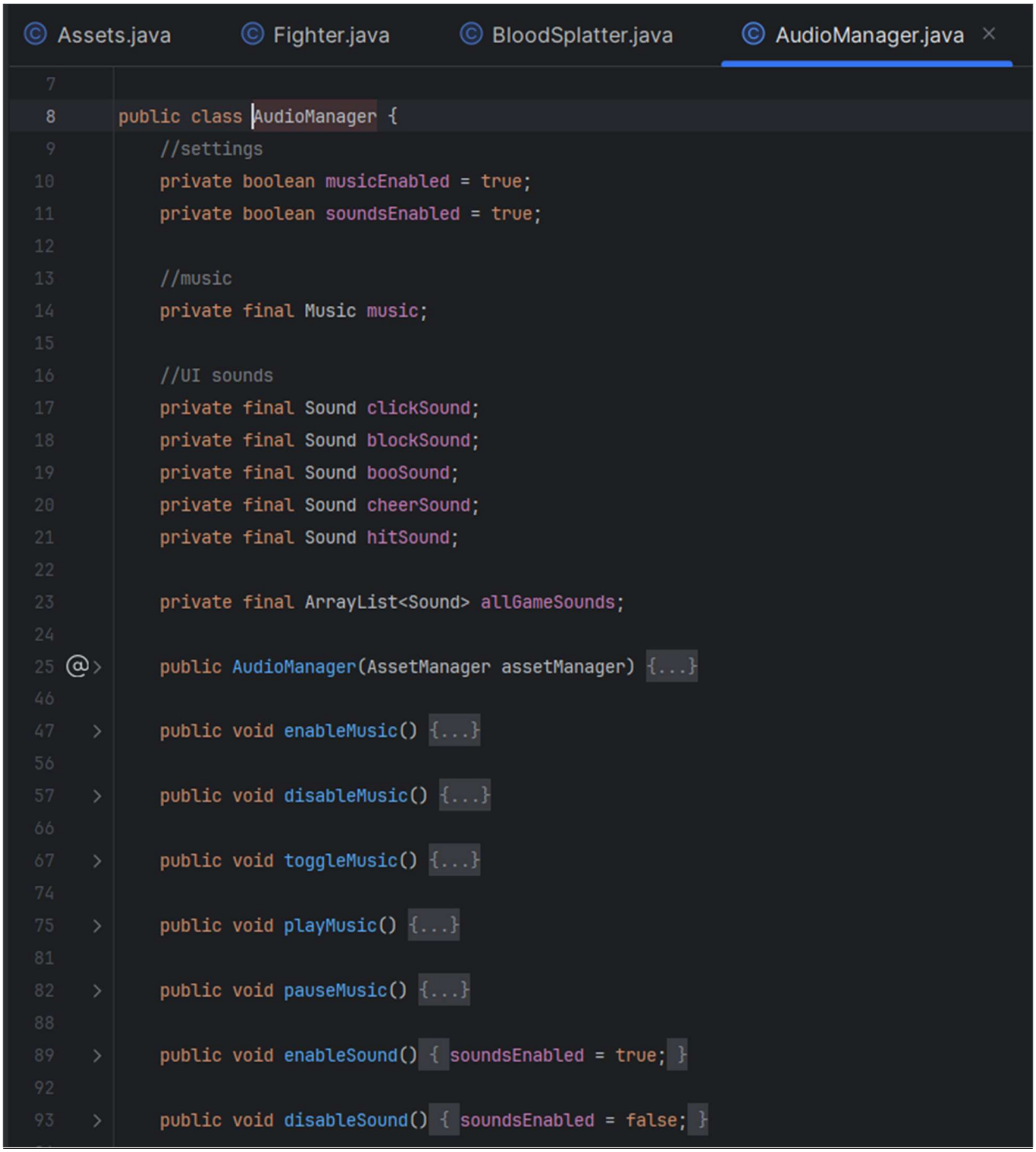
The **Assets** class **abstracts** the **AssetManager** from LibGDX, centralizing the logic for **loading** and **retrieving** game assets and keep it intact in one place.

eScreen.java | Assets.java | Fighter.java | BloodSplatter.java | Audic

```java
package com.bgrant.wwe.resources;

import ...

public class Assets {
    // gameplay assets
    public static final String BACKGROUND_TEXTURE = "textures/Background.png";
    public static final String FRONT_ROPES_TEXTURE = "textures/FrontRopes.png";
    public static final String IDLE_SPRITE_SHEET = "sprites/IdleSpriteSheet.png";
    public static final String WALK_SPRITE_SHEET = "sprites/WalkSpriteSheet.png";
    public static final String PUNCH_SPRITE_SHEET = "sprites/PunchSpriteSheet.png";
    public static final String KICK_SPRITE_SHEET = "sprites/KickSpriteSheet.png";
    public static final String HURT_SPRITE_SHEET = "sprites/HurtSpriteSheet.png";
    public static final String BLOCK_SPRITE_SHEET = "sprites/BlockSpriteSheet.png";
    public static final String WIN_SPRITE_SHEET = "sprites/WinSpriteSheet.png";
    public static final String LOSE_SPRITE_SHEET = "sprites/LoseSpriteSheet.png";
    public static final String GAMEPLAY_BUTTONS_ATLAS = "textures/GameplayButtons.atlas";
    public static final String BLOOD_ATLAS = "textures/Blood.atlas";
    // fonts
    public static final String ROBOTO_REGULAR = "fonts/Roboto-Regular.ttf";
    public static final String SMALL_FONT = "smallFont.ttf";
    public static final String MEDIUM_FONT = "mediumFont.ttf";
    public static final String LARGE_FONT = "largeFont.ttf";
    // audio assets
    public static final String BLOCK_SOUND = "audio/block.mp3";
    public static final String BOO_SOUND = "audio/boo.mp3";
    public static final String CHEER_SOUND = "audio/cheer.mp3";
    public static final String CLICK_SOUND = "audio/click.mp3";
    public static final String HIT_SOUND = "audio/hit.mp3";
    public static final String MUSIC = "audio/music.ogg";
    // menu assets
    public static final String MENU_ITEMS_ATLAS = "textures/MenuItems.atlas";
    // asset manager
    public final AssetManager manager = new AssetManager();
```

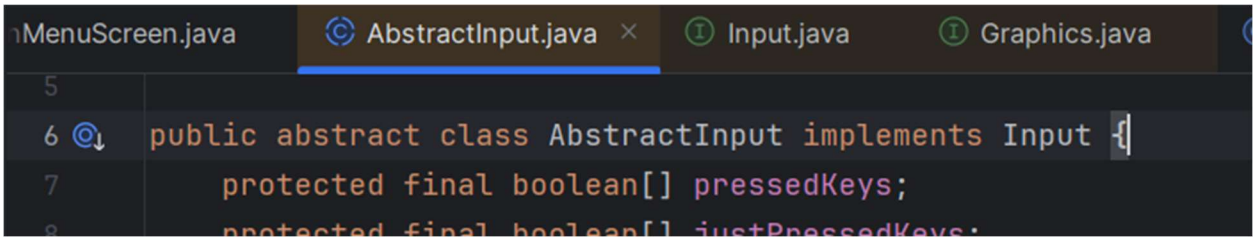
AudioManager

The **AudioManager** class **abstracts sound** and **music playback**, providing easy-to-use methods for playing **sounds** and **music** .



AbstractInput

AbstractInput provides a **base class** for handling input. It defines the structure for storing the **state** of **keys** and checking if they are **pressed**.



Conclusion

Through the use of **abstraction**, the game code becomes more **modular** and **maintainable**. Key aspects of the game, such as game **objects, resources, screens, and user input**, are all **abstracted into clear and concise classes**. This approach fosters greater flexibility in extending and modifying the game, as **new implementations** (e.g., different input devices or resources) can be **introduced without altering** the core **game logic**.

- **Game objects :** like **Fighter** and **BloodPool** abstract away their specific implementations, focusing on common behaviors and allowing for easy modification of combat mechanics.
- **Resource management :** through **Assets** and **AudioManager** simplifies access to game **resources**, ensuring that the game can easily load and manage assets such as **textures**, **sounds**, and **music**.
- **Screens :** implement a **uniform interface**, enabling **smooth transitions** between different game states, like the **main menu, gameplay**, and **settings**, without complex logic.
- **Input handling :** is abstracted into flexible classes like **AbstractInput and KeyboardInput**, allowing for easy extension to **support different input devices**, such as gamepads or touchscreens.

ENCAPSULATION

Encapsulation is the practice of bundling data (fields) and methods (behaviors) into a single unit (a class) to create self-contained, modular components.

How Encapsulation Was Used

1. **Data & Logic Bundling**
    - ***Private Fields:*** Variables like life, state, and **movementDirection** were kept private inside the Fighter class.
    - ***Public Methods:*** Actions like **moveLeft**(), **getHit**(), and **update**() operated on those fields.
2. **Controlled Access**
    - ***Getters:*** getLife() allowed read-only access to life.
    - ***Setters/Actions:*** Methods like **getHit**() enforced rules, such as blocking damage.

How Encapsulation Helped

- **Data Integrity:** Prevented invalid values and kept state changes consistent.
- **Security:** Protected key fields from outside interference.
- **Maintainability:** Internal changes didn't break external code.
- **Flexibility:** New features (like a "stun" state) only needed changes inside the class.

All of the classes and methods in the game follow this same pattern, ensuring consistency across the codebase.

Fighter Class : Private Attributes

GameScreen.java | Assets.java | Fighter.java | BloodSplatter.java

```
14    public class Fighter {
45
46        private State state;
47        private float stateTime;
48        private State renderState;
49        private float renderStateTime;
50        private final Vector2 position = new Vector2();
51        private final Vector2 movementDirection = new Vector2();
52        private float life;
53        private int facing;
54        private boolean madeContact;
```

Fighter Class : Public Methods

```
public class Fighter {

    public Vector2 getPosition() { return position; }
    public float getLife() { return life; }
    public void getReady(float positionX, float positionY) {...}
    public void render(SpriteBatch batch) {...}
    public void update(float deltaTime) {...}
    public void faceLeft() { facing = -1; }
    public void faceRight() { facing = 1; }
    private void changeState(State newState) {...}
    private void setMovement(float x, float y) {...}
    public void moveLeft() { setMovement(-1, movementDirection.y);
    public void moveRight() { setMovement(1, movementDirection.y);
    public void moveUp() { setMovement(movementDirection.x, 1); }
    public void moveDown() { setMovement(movementDirection.x, -1);
```

INFORMATION HIDING

Information hiding ensures the internal state of an object is protected from unintended interference, exposing only essential operations to maintain data integrity and simplify interactions.

*****CODE SNIPPETS*****

Private Method:

sktopLauncher.java | Fighter.java | SFS.java | GameScree

```
public class Fighter {

    private void setMovement(float x, float y) {
        movementDirection.set(x, y);
        if (state == State.WALK && x == 0 && y == 0) {
            changeState(State.IDLE);
        } else if (state == State.IDLE && (x != 0 || y != 0)) {
            changeState(State.WALK);
        }
    }
```

Public Methods:

```
    public void moveLeft() { setMovement(-1, movementDirection.y); }

    public void moveRight() { setMovement(1, movementDirection.y); }

    public void moveUp() { setMovement(movementDirection.x, 1); }

    public void moveDown() { setMovement(movementDirection.x, -1); }

    public void stopMovingLeft() {...}

    public void stopMovingRight() {...}

    public void stopMovingUp() {...}

    public void stopMovingDown() {...}
```

How Information Hiding Works ?

1. **Hidden Logic**
    - **setMovement() is private, hiding how movement input affects state (e.g., switching to WALK/IDLE).**
    - **External classes like GameScreen use simple methods like moveLeft() without knowing the internal details.**
2. **Controlled Access**
    - **Public methods expose clean actions, while fields like state and movementDirection are private to avoid invalid changes.**
3. **Enforced Rules**
    - **setMovement() ensures proper state updates, like switching to IDLE when movement stops.**

Why It Matters ?

- **Security:** Blocks invalid state changes from outside**.**
- **Simplicity:** Keeps the interface clean and intuitive.
- **Maintainability:** Only setMovement() needs updates for new features like diagonal movement.

By hiding complexity, the class stays reliable and easier to manage.

POLYMORPHISM

Polymorphism is a key concept in **Object-Oriented Programming** (OOP) and refers to the **ability** of an object to take on **many forms**. It enables the use of a **single symbol** to **represent multiple different types** and provides a **single interface** to **entities of different types**. Polymorphism is more about **interface** (**subtyping**) rather than **implementation** (**inheritance**), allowing different classes to provide their own specific implementations of the same methods.

In **game development**, polymorphism allows us to create flexible and scalable code by treating different objects, such as various fighter types or enemies, in a uniform way. This can simplify programming by enabling us to "**program in the general**" rather than "**program in the specific**." Through polymorphism, you can write programs that process objects sharing the same superclass as if they were all objects of that superclass. This allows for easy **handling of different types of characters** in a game while keeping the system **flexible** for **future extensions**.

Why Polymorphism is Used

Polymorphism is used to allow objects of different types to interact in a unified way. In the **context of the Fighter class**, polymorphism allows us to **treat all fighter characters as instances of a common superclass Fighter**. This simplifies code by enabling us to use the **same method** across different **fighter types**, with each type providing its own specific implementation of the method.

This **flexibility** helps reduce **code duplication** and promotes the reuse of code, as we don't need to **create separate methods** for each **fighter class**. Instead, we rely on **polymorphism** to define behaviors that differ across various types but share the **same method signature**.

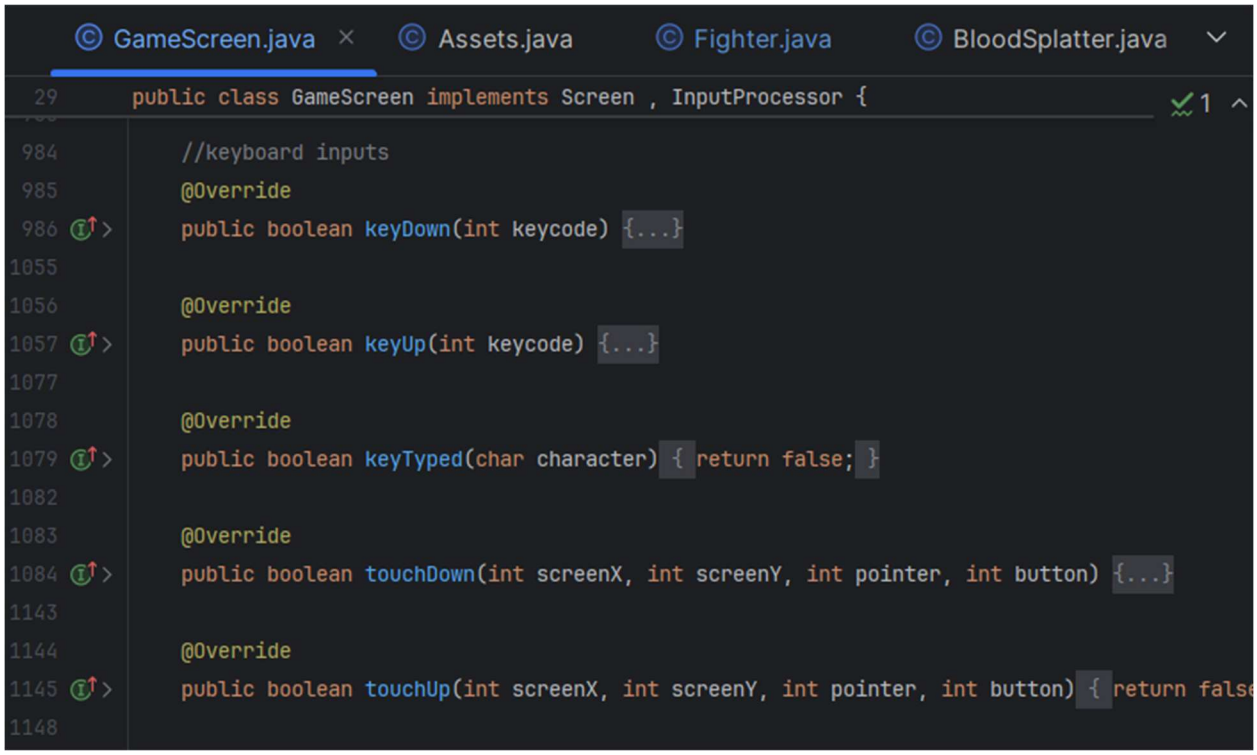
CODE SNIPPETS

1. Method Overriding (Runtime Polymorphism)

```
public class GameScreen implements Screen , InputProcessor {
    // MAIN RENDER GAME
    @Override
    public void render(float delta) {
        ScreenUtils.clear(0, 0, 0, 1);

        //update the game -- delta time should be 0 if the game isn't running ,
        update(gameState == GameState.RUNNING ? delta : 0f);

        //set the sprite batch and the shape renderer to use the viewports camer
        game.batch.setProjectionMatrix(viewport.getCamera().combined);
        game.shapeRenderer.setProjectionMatrix(viewport.getCamera().combined);

        //begin drawing
        game.batch.begin();

        //draw the background
        game.batch.draw(backgroundTexture, 0, 0, backgroundTexture.getWidth() *
            backgroundTexture.getHeight() * GlobalVariables.WORLD_SCALE);

        //draw  the blood pools
        renderBloodPools();

        //draw the fighters
        renderFighters();
```

- **GameScreen** overrides the **render(float)** method defined in **Screen interface**.
- At runtime, it calls **render(delta)** on the active **Screen reference,** dispatching to whichever subclass is in use.
- This lets you swap screens (**gameplay**, **menu**, **settings**) without changing the main loop—**classic runtime polymorphism**.

2. Interface-Driven Polymorphism

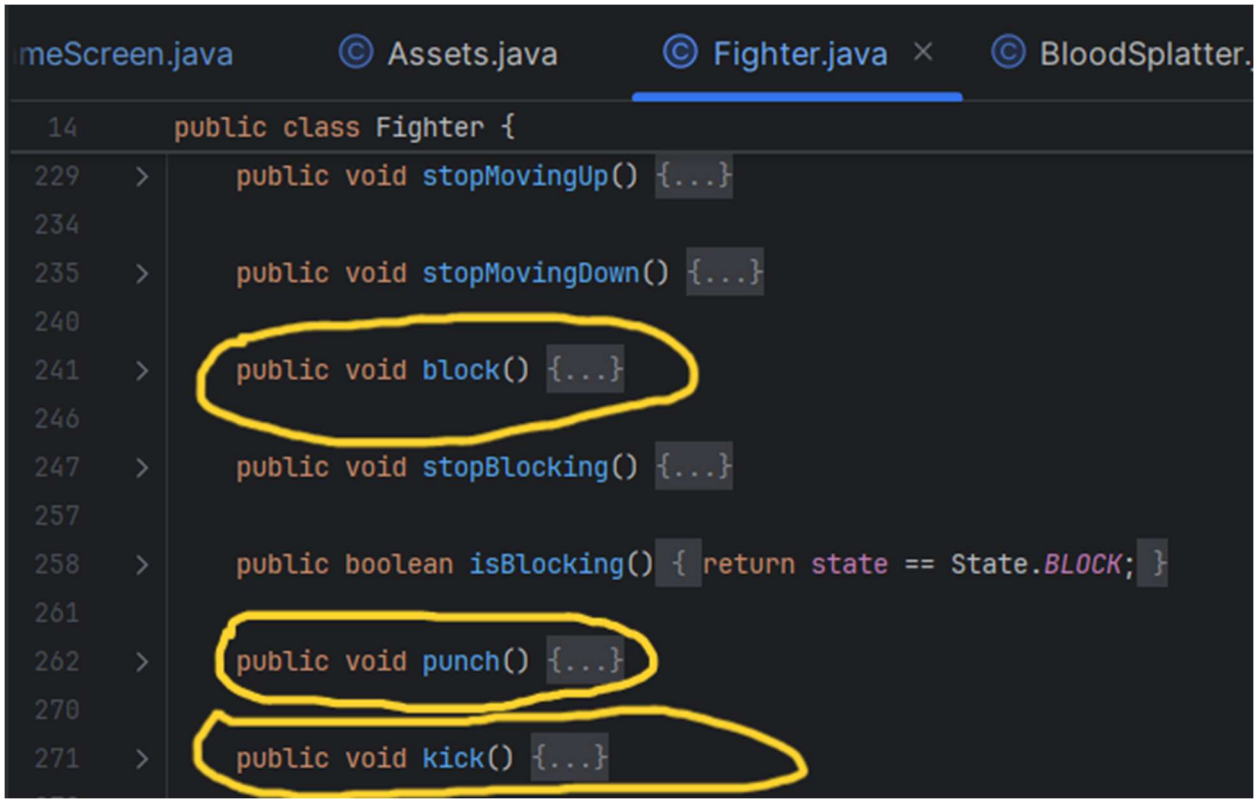


- **GameScreen implements the InputProcessor interface**, overriding methods like **keyDown** and **touchDown**.

- **At runtime**, LibGDX's input dispatcher holds your screen as an **InputProcessor** reference and invokes **keyDown()**, **touchDown()**, etc., on whatever implementing class is active.

- **Benefit:** You can **swap** out or **extend input handling** (e.g. **switch from gameplay to menu controls**) simply by changing the **InputProcessor** instance—no changes to the **input loop itself are needed**. This **is classic runtime polymorphism via interface implementation**.

3. Method Overloading (Compile-Time Polymorphism)



- **Fighter defines multiple attack() methods** with different method signatures (one with no parameters, one with a String).
- **At compile time**, the Java compiler determines which attack() method to invoke based on the argument passed in the method call.
- **This allows** the same method name to serve multiple purposes, improving code readability and enabling behavior variation without needing different method names.
- **Method overloading** allows developers to use the same method name for different behaviors based on input, improving code clarity and flexibility. It reduces the need for multiple method names and helps maintain a consistent interface, making the code easier to understand and maintain.

4.Polymorphism by Inclusion (Subtype Polymorphism)

```
    //fighters
    public Fighter player, opponent;
    public final ArrayList<FighterChoice> fighterChoiceList = new ArrayList<>();
```

```
private void loadFighterChoiceList() {
    //load the fighter choice list from assets
    Json json = new Json();
    JsonValue fighterChoices = new JsonReader().parse(Gdx.files.internal("dat

    for (int i = 0; i < fighterChoices.size; i++) {
        fighterChoiceList.add(json.fromJson(FighterChoice.class , String.valu
    }
}

@Override
```

rocessor.java | SFS.java | Game.java | FighterChoice.java | Ga

```
package com.bgrant.wwe.objects;

import com.badlogic.gdx.graphics.Color;

public class FighterChoice {
    public String name;
    public float[] colorValues;
    private Color color;

    public String getName() {
        return name;
    }

    public Color getColor() {
        if (color == null ) {
            //if the color hasn't been initialized , then do it using the color values
            color = new Color(colorValues[0] , colorValues[1] ,  colorValues[2] , 1);
        }
        return color;
    }
}
```

EXAMPLE 1:

Use case in game: **Managing fighters in a single array**

How it works and why it's useful ?

In this example, we create an array of the superclass Fighter, which acts as a container to store different types of fighters. Even though the array is declared with the type Fighter, we can assign any object that is a subclass of Fighter.

This is possible because of **inheritance**, which allows a subclass object to be treated as an instance of its parent class.

When we loop through the array method Java automatically determines which player of the method to call based on the **actual object type**, not the reference type. This is called **dynamic method dispatch**. It allows us to handle multiple fighter types in a **single unified structure**, making the game logic more flexible and easier to manage.

***NOTE:**

A JSON FILE IS CREATED WITH DIFFERENT PLAYERS*

EXAMPLE 2 :

By-Inclusion Polymorphism (Subtype Polymorphism)

Concept:

By-inclusion polymorphism, also known as subtype polymorphism, occurs when a superclass reference (or interface type) is used to point to subclass objects. It allows you to write flexible, extensible code by treating different types as instances of a shared supertype.

Implementation in the Game:

In the **GameScreen** class, the game renders different objects—**Fighter**, **BloodPool**—each of which implements a **render(SpriteBatch batch)** method. Instead of calling **render separately** on each type, we can refactor them to implement a **shared interface**, like this.

Benefit Of Subtype polymorphism:

- **Cleaner, more modular code:** You eliminate repeated loops and method calls.
- **Flexible and extensible:** You can easily add new renderable objects (like explosions, weapons, effects) without modifying the render logic.
- **Follows OOP best practices:** Promotes abstraction and reduces coupling between game logic and rendering flow.

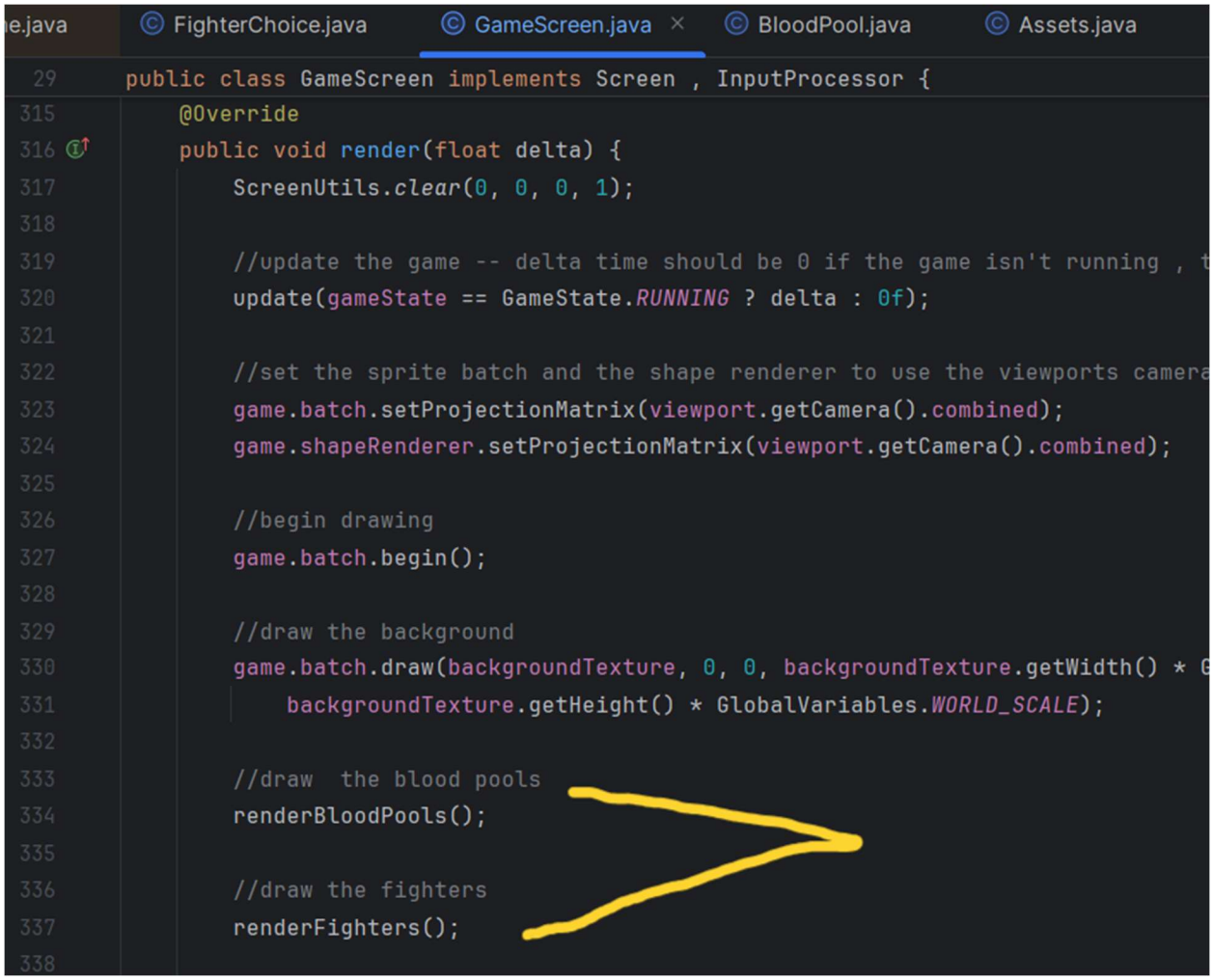
Step 1: Implement in Relevant Classes (BLOODPOOL & FIGHTER)

Game.java | FighterChoice.java | GameScreen.java | BloodPool.java | Assets.java | Fighter.java

```
ublic class BloodPool {

    public void render(SpriteBatch batch) {
        //if not active don't render
        if (!active) return;

        //set the sprite batch's color using the blood pool's alpha
        batch.setColor(1,1,1,alpha);

        //draw the blood pool
        batch.draw(texture, position.x , position.y , texture.getRegionWidth() *
                        texture.getRegionHeight() * GlobalVariables.WORLD_SCALE);

        //reset the sprite batch's color to fully opaque
        batch.setColor(1,1,1,1);
    }
```

```
    // Render method with polymorphic behavior based on fighter's state
    public void render(SpriteBatch batch) {
        //get the current animation frame
        TextureRegion currentFrame;
        switch (renderState) {
            case BLOCK:
                currentFrame = blockAnimation.getKeyFrame(renderStateTime, true);
                break;
            case HURT:
                currentFrame = hurtAnimation.getKeyFrame(renderStateTime, false);
                break;
            case IDLE:
                currentFrame = idleAnimation.getKeyFrame(renderStateTime, true);
                break;
            case KICK:
                currentFrame = kickAnimation.getKeyFrame(renderStateTime, false);
```

Step 2: Polymorphic Rendering in GameScreen



***Each of these calls is already behaving polymorphically—each class defines its own render() method, and they are treated interchangeably as long as they follow the same interface contract. ***

5.Coercion (Implicit Conversion)

```
public class Fighter {
    //number of frame rows and columns in each animation sprite sheet
    private static final int FRAME_ROWS = 2, FRAME_COLS = 3;

    //how fast the fighter can move
    public static final float MOVEMENT_SPEED = 10f;

    //maximum life a fighter can have
    public static final float MAX_LIFE = 100f;

    //amount of damage a fighter's hit will inflict
    public static final float HIT_STRENGTH = 5f;

    //factor to decrease damage if a fighter gets hits by while blocking
    public static final float BLOCK_DAMAGE_FACTOR = 0.2f;
```

Coercion Example 1: Damage Calculation getHit() method

```
    public void getHit(float damage) {
        if (state == State.HURT || state == State.WIN || state == State.LOSE) return;

        //render the fighters life by the full amount or a fraction of it if the fighter is blocking
        life -= state == State.BLOCK ? damage * BLOCK_DAMAGE_FACTOR : damage;

        if (life <= 0f) {
            //if no life remains , lose
            lose();
        } else if (state != State.BLOCK) {
            //if not blocking , go to hurt state
            changeState(State.HURT);
        }
    }
```

- In this scenario, coercion happens when the damage (a float) and **BLOCK_DAMAGE_FACTOR** (a float) are multiplied. The result is still a float, but we cast it to an int before subtracting it from life (which is now an int).
- This is an example of explicit coercion happening through the cast (int) for both the damage calculation and the subtraction from life.

Coercion Example 2: isAttackActive() Method with StateTime

```java
public boolean isAttackActive() {
    //the attack is only active if the fighter has not yet made contact and the attack animation has not just started
    //or is almost finished
    if (hasMadeContact()) {
        return false;
    } else if (state == State.PUNCH) {
        return stateTime > punchAnimation.getAnimationDuration() * 0.33f &&
            stateTime < punchAnimation.getAnimationDuration() * 0.66f;
    } else if (state == State.KICK) {
        return stateTime > kickAnimation.getAnimationDuration() * 0.33f &&
            stateTime < kickAnimation.getAnimationDuration() * 0.66f;
    } else {
        return false;
    }
}
```

- **Explanation:** stateTime and getAnimationDuration() both return float values, and the constants 0.33f and 0.66f are also floats. The multiplication of getAnimationDuration() (float) with 0.33f (float) will result in a float.
- **Coercion:** This is another case where implicit type coercion happens, since the expression deals with floating-point numbers.
- If stateTime (or getAnimationDuration()) were an integer instead of a float, the multiplication with 0.33f (float) would implicitly convert the integer into a float before performing the multiplication.

Benefit of Coercion in Game Development:
Coercion simplifies code by allowing automatic type conversions between different data types, such as when mixing integers and floats. This reduces the need for explicit casting, making the code cleaner and less error-prone, which is especially useful in fast-paced calculations like damage or health management in games.

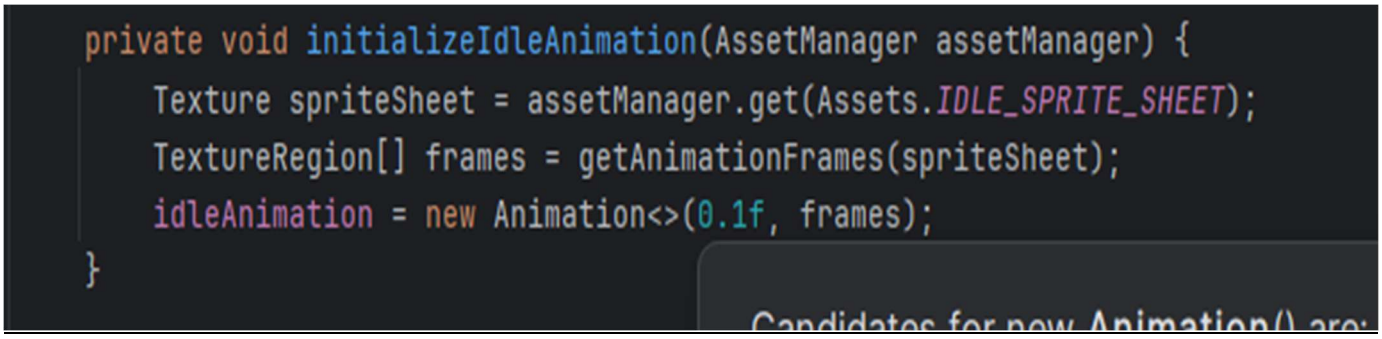
5. Parametric Polymorphism

Animation<T> class to manage **character animations**. Here's how parametric polymorphism is applied:

```
14    public class Fighter {
56        //animations
57        private Animation<TextureRegion> blockAnimation;
58        private Animation<TextureRegion> hurtAnimation;
59        private Animation<TextureRegion> idleAnimation;
60        private Animation<TextureRegion> kickAnimation;
61        private Animation<TextureRegion> loseAnimation;
62        private Animation<TextureRegion> punchAnimation;
63        private Animation<TextureRegion> walkAnimation;
64        private Animation<TextureRegion> winAnimation;
```

DID THE SAME THING FOR DIFFERENT ANIMATION

```
    private void initializeBlockAnimation(AssetManager assetManager) {...}

    private void initializeHurtAnimation(AssetManager assetManager) {...}

    private void initializeIdleAnimation(AssetManager assetManager) {...}

    private void initializeKickAnimation(AssetManager assetManager) {...}

    private void initializeLoseAnimation(AssetManager assetManager) {...}

    private void initializePunchAnimation(AssetManager assetManager) {...}

    private void initializeWalkAnimation(AssetManager assetManager) {...}

    private void initializeWinAnimation(AssetManager assetManager) {...}
```



```
private void initializeIdleAnimation(AssetManager assetManager) {
    Texture spriteSheet = assetManager.get(Assets.IDLE_SPRITE_SHEET);
    TextureRegion[] frames = getAnimationFrames(spriteSheet);
    idleAnimation = new Animation<>(0.1f, frames);
}
```

How It Works:

1. **Generic Class**: Animation<T> is defined to work with any type T (e.g., TextureRegion, Sprite).
2. **Specialization**: By declaring Animation<TextureRegion>, you tell Java to enforce that this animation only uses TextureRegion objects.
3. **Reusability**: The same Animation<T> class can handle other types (e.g., Animation<Sprite>) without code duplication.

How It Works Under the Hood

- **Type Erasure**: At compile time, Java replaces T with TextureRegion (the bound type) to enforce type checks.
- **Compiler Enforcement**: The generic type <TextureRegion> acts as a contract. The compiler validates that all method calls (e.g., getKeyFrame()) return TextureRegion objects.

Benefits of Polymorphism

1. **Code Reusability:** Polymorphism enables **code reuse** by allowing different types of objects to share a **common method interface**, reducing the need for **duplicate code**.

2. **Simplified Code:** With **polymorphism**, you can handle objects of different types in a **uniform** way. This **reduces** the **complexity** of the code, making it cleaner and easier to understand.

3. **Flexibility and Scalability:** Polymorphism enhances the flexibility of the game. New character types or behaviors can be added by simply extending the Fighter class without altering existing code. This makes the system more scalable and adaptable to future needs.

4. **Maintenance:** Since polymorphism allows you to interact with objects in a general way, it simplifies code maintenance. You can make changes to the superclass or subclass without needing to modify multiple methods in different parts of the program.

Conclusion

Polymorphism is a powerful OOP concept that allows objects of different types to be treated uniformly, simplifying code management and enabling greater flexibility and scalability. In our game, **polymorphism facilitates handling multiple fighter types** with a unified interface, ensuring that the game logic remains **clear** and **adaptable**. By applying **polymorphism**, we can extend the game with new **features** and character types without significantly altering the existing code. As with other key OOP principles, polymorphism **plays a critical role** in making the game **both efficient and easy** to maintain. All classes in the game follow similar polymorphic patterns, **ensuring consistent functionality** and **seamless interaction** between objects.

References

Professor's Slides and Lectures :

The foundational understanding of **Object-Oriented Programming (OOP)** concepts, such as **inheritance, abstraction, encapsulation, and polymorphism**, was primarily derived from the **lectures** and **slides** provided by my **professor**. These materials laid the groundwork for implementing OOP principles in my game project.

LibGDX Courses & Tutorials

1. Udemy Courses on LibGDX

I completed several LibGDX courses on Udemy, covering game mechanics, animations, and OOP principles for 2D games.
Link: https://www.udemy.com

2. YouTube - LibGDX Tutorials

A range of LibGDX tutorials on YouTube, from basic setup to advanced topics like shaders and networking.
Link: https://www.youtube.com/results?search_query=libgdx+tutorial

Resources & Assets

1.A few of the **assets**, such as **custom UI elements**, **character variations**, and **logo designs**, were **created by me** to add a personal and original touch to the game.

2.Graphics & Sprites - OpenGameArt.org and Itch.io

3. Sound Effects & Music - Freesound.org and Kenney.nl

GitHub Repository

The complete source code for the **Wrestle Mania (WWE)** game, along with additional resources, can be accessed via my **GitHub repository**. This contains all the files, assets, and implementations used in the development of the game.
Link: https://github.com/anshpandey2004/WWE