Präsentatioon

Main File

Marcel:
private GameManager gameManager; -> Instanzvariable von GameManager

start Methode:

- gameManager = new GameManager(primaryStage); -> übergibt dem gamemanager die primary Stage (damit er die Szenen machen kann)
- primaryStage.setTitle("2D Fighting Game"); -> macht Fenstertitel
- primaryStage.setScene(new Scene(gameManager.getMainMenu(), 800, 600)); -> macht die Stage in einer bestimmten Größe (hier 800x600)
- primaryStage.show(); -> zeigt den Schaß an

main Methode: launch(args); -> ruft die start methode auf

GameArena File

Sophie:

Diese Methode erstellt die Spielarena und initialisiert alle wichtigen Elemente wie Spieler, Boden, Gesundheitsbalken und Steuerung.

```
public GameArena(String p1Type, String p2Type, GameManager
gameManager) {
```

- **Funktion**: Konstruktor der Klasse GameArena. Er erstellt die Arena und initialisiert die Spieler und UI-Elemente.
- **Parameter**:
  - p1Type: Der Charaktertyp von Spieler 1.
  - p2Type: Der Charaktertyp von Spieler 2.
  - gameManager: Referenz auf den GameManager, um später auf Spielinformationen zuzugreifen.

```
this.gameManager = gameManager;
```

- Speichert die Referenz auf den GameManager, um später auf Spielerinformationen und andere Spielsteuerungen zugreifen zu können.

```
gamePane = new Pane();
gamePane.setPrefSize(800, 600);
```

- Erstellt ein neues Pane als Container für alle visuellen Elemente der Arena.
- Setzt die Größe der Arena auf 800x600 Pixel.

```
activeKeys = new HashSet<>();
```

- Initialisiert eine Set-Datenstruktur, um die aktuell gedrückten Tasten zu speichern.

```
Rectangle floor = new Rectangle(0, 500, 800, 100);
floor.setFill(Color.GRAY);
```

- Erstellt ein Rechteck, das den Boden der Arena darstellt.
- Der Boden beginnt bei x=0, y=500, hat eine Breite von 800 und eine Höhe von 100.
- Färbt den Boden grau.

```
player1 = new Character(p1Type, 200, 300);
player2 = new Character(p2Type, 600, 300);
```

- Erstellt die beiden Spielercharaktere mit den übergebenen Typen.
- Spieler 1 wird bei x=200, y=300 positioniert, Spieler 2 bei x=600, y=300.

```
setupHealthBars();
setupControls();
startGameLoop();
```

- Ruft drei Methoden auf:

- setupHealthBars(): Erstellt die Gesundheitsbalken für beide Spieler.
- setupControls(): Konfiguriert die Steuerung für die Spieler.
- startGameLoop(): Startet den Haupt-Gameloop, der das Spiel kontinuierlich aktualisiert.

```
gamePane.getChildren().addAll(floor, player1, player2, healthBar1,
healthBar2, healthLabel1, healthLabel2);
```

- Fügt den Boden, die Spieler, die Gesundheitsbalken und die Gesundheitslabels zum Pane hinzu, damit sie in der Arena sichtbar sind.

```
player1StatsLabel = new Label(String.format("%s - Wins: %d",
gameManager.getPlayer1Name(),
PlayerData.getWins(gameManager.getPlayer1Name())));

player2StatsLabel = new Label(String.format("%s - Wins: %d",
gameManager.getPlayer2Name(),
PlayerData.getWins(gameManager.getPlayer2Name())));
```

- Erstellt Labels, die die Namen und Siege der Spieler anzeigen.
- Die Siege werden aus der PlayerData-Klasse geladen.

```
player1StatsLabel.setLayoutX(50);
player1StatsLabel.setLayoutY(60);
player2StatsLabel.setLayoutX(550);
player2StatsLabel.setLayoutY(60);
```

- Positioniert die Statistik-Labels für Spieler 1 und Spieler 2 in der Arena.

```
gamePane.getChildren().addAll(player1StatsLabel, player2StatsLabel);
```

- Fügt die Statistik-Labels zum Pane hinzu, damit sie sichtbar sind.

---

2. setupHealthBars()

Diese Methode erstellt die Gesundheitsbalken und Labels für beide Spieler.

```
private void setupHealthBars() {
```

- **Funktion**: Initialisiert die Gesundheitsbalken und Labels für beide Spieler.

```
healthBar1 = new Rectangle(50, 20, 200, 20);
healthBar2 = new Rectangle(550, 20, 200, 20);
```

- Erstellt zwei grüne Rechtecke, die die Gesundheitsbalken darstellen.
- healthBar1 ist für Spieler 1 und wird bei x=50, y=20 positioniert.
- healthBar2 ist für Spieler 2 und wird bei x=550, y=20 positioniert.

```
healthBar1.setFill(Color.GREEN);
healthBar2.setFill(Color.GREEN);
```

- Färbt beide Gesundheitsbalken grün.

```
healthLabel1 = new Label("100/100");
healthLabel2 = new Label("100/100");
```

- Erstellt zwei Labels, die die aktuellen und maximalen Lebenspunkte der Spieler anzeigen.

```
healthLabel1.setLayoutX(50);
healthLabel1.setLayoutY(40);
healthLabel2.setLayoutX(550);
healthLabel2.setLayoutY(40);
```

- Positioniert die Gesundheitslabels direkt über den Gesundheitsbalken.

---

3. setupControls()

Diese Methode konfiguriert die Steuerung für die Spieler.

```
private void setupControls() {
```

- **Funktion**: Initialisiert die Tastatureingaben für die Spieler.

```
gamePane.setFocusTraversable(true);
gamePane.requestFocus();
```

- Aktiviert die Fokussierbarkeit des Pane, damit es Tastatureingaben empfangen kann.
- Fordert den Fokus an, damit das Pane sofort auf Eingaben reagiert.

```
gamePane.setOnKeyPressed(e -> {
    activeKeys.add(e.getCode());
    System.out.println("Key pressed: " + e.getCode()); // Debug-
Ausgabe
    e.consume();
});
```

- Fügt gedrückte Tasten zur activeKeys-Liste hinzu.
- Gibt die gedrückte Taste in der Konsole aus (Debugging).
- Verhindert, dass das Event weiterverarbeitet wird (e.consume()).

```
gamePane.setOnKeyReleased(e -> {
    activeKeys.remove(e.getCode());
    e.consume();
});
```

- Entfernt losgelassene Tasten aus der activeKeys-Liste.
- Verhindert, dass das Event weiterverarbeitet wird (e.consume()).

`handleAttack(Character attacker, Character target, boolean isStrongAttack)` **

Verarbeitet Angriffe zwischen zwei Charakteren.

```
private void handleAttack(Character attacker, Character target,
boolean isStrongAttack) {
```

- **Funktion**: Diese Methode prüft, ob ein Angriff möglich ist, und wendet Schaden an, falls die Bedingungen erfüllt sind.
- **Parameter**:
  - `attacker` : Der angreifende Charakter.
  - `target` : Der angegriffene Charakter.
  - `isStrongAttack` : Gibt an, ob es sich um einen starken Angriff handelt.

```
if (isStrongAttack && !attacker.canUseStrongAttack()) {
    return;
}
```

- Prüft, ob der Angreifer einen starken Angriff ausführen kann. Falls nicht, wird die Methode abgebrochen.

```
if (!attacker.canAttack()) {
    return;
}
```

- Prüft, ob der Angreifer überhaupt angreifen kann (z. B. ob ein Cooldown aktiv ist). Falls nicht, wird die Methode abgebrochen.

```
boolean isRanged = attacker.getType().equals("Magician") ||
                   attacker.getType().equals("Bishop") ||
                   attacker.getType().equals("Priestess") ||
                   attacker.getType().equals("Wizard");
```

- Prüft, ob der Angreifer ein Fernkämpfer ist (z. B. Magier oder Priester).

```
if (isRanged) {
    double heightDiff = Math.abs(attacker.getY() - target.getY());
    double distance = Math.abs(attacker.getX() - target.getX());
```

- Wenn der Angreifer ein Fernkämpfer ist:
  - Berechnet die vertikale Distanz (`heightDiff`) zwischen Angreifer und Ziel.
  - Berechnet die horizontale Distanz (`distance`) zwischen Angreifer und Ziel.

```
    if (heightDiff < 50 && distance <= attacker.getAttackRange()) {
        int damage = attacker.getDamage() * (isStrongAttack ? 2 : 1);
        target.takeDamage(damage);
        attacker.setAttackCooldown();
        if (isStrongAttack) {
            attacker.setStrongAttackCooldown();
        }
    }
}
```

- Prüft, ob das Ziel in Reichweite ist (horizontal und vertikal).
- Wenn ja:
  - Berechnet den Schaden (doppelt so hoch bei einem starken Angriff).
  - Wendet den Schaden auf das Ziel an.
  - Setzt den Angriffscooldown des Angreifers.
  - Setzt den Cooldown für starke Angriffe, falls es ein starker Angriff war.

```
else {
    if (attacker.attack(target, isStrongAttack) && isStrongAttack) {
        attacker.setStrongAttackCooldown();
    }
}
}
```

- Wenn der Angreifer ein Nahkämpfer ist:
  - Führt den Angriff aus.
  - Setzt den Cooldown für starke Angriffe, falls es ein starker Angriff war.

---

2. `update()`

Die zentrale Methode, die in jedem Frame aufgerufen wird.

```
private void update() {
    handleMovement();
    handleAttacks();
    updatePhysics();
}
```

- **Funktion**: Diese Methode koordiniert die Bewegung, Angriffe und Physik des Spiels.
- **Ablauf**:
  - `handleMovement()`: Verarbeitet die Bewegungseingaben der Spieler.
  - `handleAttacks()`: Verarbeitet die Angriffseingaben der Spieler.
  - `updatePhysics()`: Aktualisiert die Physik (z. B. Schwerkraft, Kollisionen).

---

3. `handleMovement()`

Verarbeitet die Bewegungseingaben der Spieler.

```
private void handleMovement() {
```

- **Funktion**: Diese Methode prüft, welche Tasten gedrückt sind, und bewegt die Spieler entsprechend.

```
if (activeKeys.contains(KeyCode.A)) {
    player1.moveLeft();
}
```

- Wenn die Taste `A` gedrückt ist, bewegt sich Spieler 1 nach links.

```
if (activeKeys.contains(KeyCode.D)) {
    player1.moveRight();
}
```

- Wenn die Taste `D` gedrückt ist, bewegt sich Spieler 1 nach rechts.

```
if (activeKeys.contains(KeyCode.W)) {
    player1.jump();
}
```

- Wenn die Taste `W` gedrückt ist, springt Spieler 1.

```
if (activeKeys.contains(KeyCode.LEFT)) {
    player2.moveLeft();
}
```

- Wenn die Pfeiltaste `Links` gedrückt ist, bewegt sich Spieler 2 nach links.

```
if (activeKeys.contains(KeyCode.RIGHT)) {
    player2.moveRight();
}
```

- Wenn die Pfeiltaste `Rechts` gedrückt ist, bewegt sich Spieler 2 nach rechts.

```
if (activeKeys.contains(KeyCode.UP)) {
    player2.jump();
}
```

- Wenn die Pfeiltaste `Oben` gedrückt ist, springt Spieler 2.

```
}
```

- Ende der Methode.

Hier ist der formatierte Text mit korrekter Markdown-Formatierung für Codeblöcke:

`handleAttacks()`

Verarbeitet die Angriffseingaben der Spieler.

```
private void handleAttacks() {
```

- **Funktion**: Diese Methode prüft, ob Angriffstasten gedrückt sind, und führt Angriffe aus.

```
// Player 1 attacks
if (activeKeys.contains(KeyCode.Q) || activeKeys.contains(KeyCode.E))
{
    handleAttack(player1, player2, activeKeys.contains(KeyCode.E));
}
```

- Wenn Spieler 1 die Taste `Q` (normaler Angriff) oder `E` (starker Angriff) drückt:
- Führt die Methode `handleAttack` aus, wobei `player1` der Angreifer und `player2` das Ziel ist.
- Der Parameter `activeKeys.contains(KeyCode.E)` gibt an, ob es sich um einen starken Angriff handelt.

```
// Player 2 attacks
if (activeKeys.contains(KeyCode.K) || activeKeys.contains(KeyCode.L))
{
    handleAttack(player2, player1, activeKeys.contains(KeyCode.L));
}
```

- Wenn Spieler 2 die Taste `K` (normaler Angriff) oder `L` (starker Angriff) drückt:
- Führt die Methode `handleAttack` aus, wobei `player2` der Angreifer und `player1` das Ziel ist.
- Der Parameter `activeKeys.contains(KeyCode.L)` gibt an, ob es sich um einen starken Angriff handelt.

```
}
```

- Ende der Methode.

`updatePhysics()`

Aktualisiert die Physik des Spiels.

```
private void updatePhysics() {
```

- **Funktion**: Diese Methode aktualisiert die Physik der Spieler und prüft den Spielstatus.

```
player1.update();
player2.update();
```

- Ruft die `update()`-Methode der Spieler auf, um deren Physik (z. B. Schwerkraft, Sprünge) zu aktualisieren.

```
updateHealthBars();
```

- Aktualisiert die Gesundheitsanzeigen der Spieler.

```
checkGameOver();
```

- Prüft, ob das Spiel vorbei ist, und zeigt gegebenenfalls den Game-Over-Bildschirm an.

```
}
```

- Ende der Methode.

`updateHealthBars()`

Aktualisiert die Gesundheitsanzeigen der Spieler.

```
private void updateHealthBars() {
```

- **Funktion**: Passt die Breite der Gesundheitsbalken und die Labels an die aktuelle Gesundheit der Spieler an.

```
healthBar1.setWidth(Math.max(0, player1.getHealth() * 2));
```

- Setzt die Breite des Gesundheitsbalkens von Spieler 1 basierend auf seiner aktuellen Gesundheit.
- Die Breite wird auf `0` gesetzt, wenn die Gesundheit negativ ist.

```
healthBar2.setWidth(Math.max(0, player2.getHealth() * 2));
```

- Setzt die Breite des Gesundheitsbalkens von Spieler 2 basierend auf seiner aktuellen Gesundheit.

```
healthLabel1.setText(player1.getHealth() + "/100");
```

- Aktualisiert das Label von Spieler 1, um die aktuelle und maximale Gesundheit anzuzeigen.

```
healthLabel2.setText(player2.getHealth() + "/100");
```

- Aktualisiert das Label von Spieler 2, um die aktuelle und maximale Gesundheit anzuzeigen.

```
}
```

- Ende der Methode.

Hier ist der formatierte Text mit korrekter Markdown-Formatierung für Codeblöcke:

`checkGameOver()`

Prüft, ob ein Spieler gewonnen hat, und zeigt den Game-Over-Bildschirm an.

```
private void checkGameOver() {
```

- **Funktion**: Diese Methode prüft, ob die Gesundheit eines Spielers auf 0 gefallen ist, und beendet das Spiel, falls dies der Fall ist.

```
if (!gameOver && (player1.getHealth() <= 0 || player2.getHealth() <=
0)) {
```

- Prüft, ob das Spiel noch nicht vorbei ist (`!gameOver`) und ob die Gesundheit eines Spielers auf 0 oder weniger gefallen ist.

```
gameOver = true;
```

- Markiert das Spiel als beendet.

```
boolean isPlayer1Winner = player2.getHealth() <= 0;
```

- Bestimmt, ob Spieler 1 gewonnen hat. Dies ist der Fall, wenn die Gesundheit von Spieler 2 auf 0 gefallen ist.

```
String winnerName = isPlayer1Winner ?
    gameManager.getPlayer1Name() :
    gameManager.getPlayer2Name();
```

- Speichert den Namen des Gewinners basierend auf dem Ergebnis der vorherigen Bedingung.

```
PlayerData.addWin(winnerName);
```

- Aktualisiert die Siegesstatistiken des Gewinners in der `PlayerData` -Klasse.

```
String winner = player1.getHealth() <= 0 ? "Player 2" : "Player 1";
```

- Bestimmt, welcher Spieler gewonnen hat, und speichert dies als String.

```
Label gameOverLabel = new Label(winner + " wins!");
gameOverLabel.setStyle("-fx-font-size: 24;");
gameOverLabel.setLayoutX(350);
gameOverLabel.setLayoutY(250);
```

- Erstellt ein Label, das den Gewinner anzeigt, und positioniert es in der Mitte der Arena.

```
HBox buttons = new HBox(20);
buttons.setLayoutX(300);
buttons.setLayoutY(300);
```

- Erstellt eine horizontale Box (`HBox`) für die Buttons und positioniert sie unterhalb des Game-Over-Labels.

```
Button restartButton = new Button("Play Again");
restartButton.setOnAction(e -> restartGame());
```

- Erstellt einen Button, um das Spiel neu zu starten, und verknüpft ihn mit der Methode `restartGame` .

```
Button characterSelectButton = new Button("Character Select");
characterSelectButton.setOnAction(e ->
gameManager.showCharacterSelect());
```

- Erstellt einen Button, um zur Charakterauswahl zurückzukehren, und verknüpft ihn mit der Methode `showCharacterSelect` des `GameManager` .

```
buttons.getChildren().addAll(restartButton, characterSelectButton);
```

- Fügt die beiden Buttons zur `HBox` hinzu.

```
gamePane.getChildren().addAll(gameOverLabel, buttons);
```

- Fügt das Game-Over-Label und die Buttons zur Arena hinzu, damit sie sichtbar sind.

```
}
```

- Ende der Methode.

`restartGame()`

Setzt die Arena für eine neue Runde zurück.

```
private void restartGame() {
```

- **Funktion**: Startet das Spiel neu, indem es die Spieler zurücksetzt und alle Game-Over-Elemente entfernt.

```
player1.reset();
player2.reset();
gameOver = false;
```

- Setzt die Spieler zurück (z. B. Gesundheit und Position).
- Markiert das Spiel als nicht mehr vorbei.

```
gamePane.getChildren().removeIf(node ->
    (node instanceof Label && ((Label)
node).getText().contains("wins")) ||
    (node instanceof HBox)
);
```

- Entfernt alle Game-Over-Elemente aus der Arena:
- Labels, die "wins" enthalten.
- Buttons (in einem `HBox`).

```
player1StatsLabel.setText(String.format("%s - Wins: %d",
    gameManager.getPlayer1Name(),
    PlayerData.getWins(gameManager.getPlayer1Name())));
```

- Aktualisiert das Statistik-Label von Spieler 1 mit dem aktuellen Siegestand.

```
player2StatsLabel.setText(String.format("%s - Wins: %d",
    gameManager.getPlayer2Name(),
    PlayerData.getWins(gameManager.getPlayer2Name())));
```

- Aktualisiert das Statistik-Label von Spieler 2 mit dem aktuellen Siegestand.

```
updateHealthBars();
```

- Aktualisiert die Gesundheitsanzeigen der Spieler.

```
gamePane.requestFocus();
```

- Setzt den Fokus auf die Arena, damit Eingaben wieder erkannt werden.

```
}
```

- Ende der Methode.

`getGamePane()`

Gibt den Haupt-Spielcontainer zurück.

```
public Pane getGamePane() {
    return gamePane;
}
```

- **Funktion**: Gibt das `Pane` zurück, das die gesamte Arena enthält.

- **Verwendung**: Wird vom `GameManager` verwendet, um die Szene zu wechseln.

`reset(String p1Character, String p2Character)`

Setzt die Arena mit neuen Charakteren zurück.

```
public void reset(String p1Character, String p2Character) {
```

- **Funktion**: Entfernt die alten Spieler, erstellt neue Charaktere und fügt sie zur Arena hinzu.

```
gamePane.getChildren().removeAll(player1, player2);
```

- Entfernt die alten Spieler aus der Arena.

```
player1 = new Character(p1Character, 100, 450);
player2 = new Character(p2Character, 600, 450);
```

- Erstellt neue Charaktere mit den übergebenen Typen und setzt ihre Startpositionen.

```
gamePane.getChildren().addAll(player1, player2);
```

- Fügt die neuen Spieler zur Arena hinzu.

```
gameOver = false;
```

- Markiert das Spiel als nicht mehr vorbei.

```
updateHealthBars();
```

- Aktualisiert die Gesundheitsanzeigen der Spieler.

```
gamePane.requestFocus();
```

- Setzt den Fokus auf die Arena, damit Eingaben wieder erkannt werden.

```
}
```

- Ende der Methode.

Character File

````
# Character.java – Zeilenweise Erklärung

Die `Character`-Klasse repräsentiert eine Spielfigur und enthält alle logischen und grafischen Eigenschaften, die im Spiel verwendet werden.

---

```java
public class Character extends ImageView {
````

- Die Klasse `Character` erweitert `ImageView`, wodurch sie in der JavaFX-GUI ein Bild anzeigen kann.

```
private static final Map<String, Image> characterImages = new HashMap<>();
```

- Ein statischer Cache für Charakterbilder, um wiederholtes Laden aus dem Dateisystem zu vermeiden.

```

```
private static final String IMAGE_PATH = "/images/%s.png";
```

- Vorlage für den Pfad zum Bild eines Charakters. `%s` wird durch den Charakternamen ersetzt.

```
public static final Map<String, CharacterStats> CHARACTER_STATS =
Map.of(
    "Bishop", new CharacterStats(15, 2, 180, 12, 6, "Healer with
moderate ranged attacks"),
    "Holyknight", new CharacterStats(22, 3, 45, 18, 4, "Holy warrior
with high defense"),
    "Knight", new CharacterStats(20, 2, 45, 16, 5, "Well-armored
fighter with balanced stats"),
    "Magician", new CharacterStats(25, 2, 200, 5, 4, "Powerful
spellcaster with high damage"),
    "Ninja", new CharacterStats(14, 5, 35, 6, 9, "Fastest character
with rapid attacks"),
    "Priestess", new CharacterStats(18, 2, 160, 8, 7, "Support caster
with healing abilities"),
    "Rogue", new CharacterStats(16, 4, 35, 6, 8, "Quick melee fighter
with high attack speed"),
    "Swordsman", new CharacterStats(19, 3, 40, 12, 6, "Balanced
fighter with good reach"),
    "Warrior", new CharacterStats(21, 3, 40, 15, 5, "Strong melee
fighter with good defense"),
    "Wizard", new CharacterStats(28, 2, 220, 4, 3, "Master of
destructive magic")
);
```

- Enthält zu jedem Charaktertyp ein `CharacterStats`-Objekt mit den zugehörigen Werten:
  - `damage`, `speed`, `range`, `defense`, `attackSpeed`, `description`.

```
static {
    for (String type : CHARACTER_STATS.keySet()) {
        try {
            String imagePath = String.format(IMAGE_PATH,
type.toLowerCase());
            Image img = new Image(
                Character.class.getResource(imagePath).toString(),
                450,
                450,
                true,
                true
            );
            characterImages.put(type, img);
        } catch (Exception e) {
            System.err.println("Error loading image for " + type + ":
" + e.getMessage());
        }
    }
}
```

- Ein statischer Block, der beim Programmstart ausgeführt wird.
- Lädt für jeden Charakter das zugehörige Bild (450x450, skaliert proportional) und speichert es im `characterImages`-Cache.
- Bei Fehlern wird eine Meldung in die Konsole geschrieben.

```
private String type;
```

- Der Typ des Charakters (z. B. "Ninja", "Wizard").

```
private int health = 100;
private int maxHealth = 100;
```

- Aktuelle und maximale Lebenspunkte des Charakters.

```
private int damage;
private double speed;
```

- Der verursachte Schaden und die Bewegungsgeschwindigkeit.

```
private boolean canAttack = true;
private long lastAttackTime;
private static final int ATTACK_COOLDOWN = 500;
```

- Steuerung für einfache Angriffe:
  - Ob man angreifen darf (`canAttack`)
  - Zeitpunkt des letzten Angriffs
  - Cooldown in Millisekunden (0.5 Sekunden)

```
private double velocityY = 0;
private boolean isJumping = false;
```

- Vertikale Geschwindigkeit (für Sprünge) und Sprungstatus.

```
private double attackRange;
private int defense;
private int attackSpeed;
private String description;
```

- Weitere Kampfwerte und Beschreibung (aus `CharacterStats`).

```
private boolean facingRight = true;
```

- Gibt an, ob der Charakter nach rechts schaut (für Bilddrehung wichtig).

```
private boolean canUseStrongAttack = true;
private long lastStrongAttackTime = 0;
private static final long STRONG_ATTACK_COOLDOWN = 3000;
```

- Status und Cooldown für starke Angriffe (3 Sekunden Cooldown).

```
private static final double GRAVITY = 0.5;
private static final double JUMP_FORCE = -12;
private static final double GROUND_Y = 450;
```

- Physikwerte:
  - Schwerkraft
  - Sprungkraft (negativ für Aufwärtsbewegung)
  - Y-Position des Bodens (für Landung)

```
public Character(String type, double x, double y) {
```

- Konstruktor: Erstellt einen neuen Charakter an der Position `(x, y)` mit dem angegebenen Typ.

```
    super(characterImages.getOrDefault(type,
characterImages.get("Warrior")));
```

- Ruft den Konstruktor von `ImageView` auf und setzt das Bild des Charakters.
- Falls kein Bild für den Typ gefunden wurde, wird das Bild des „Warrior“ verwendet.

```
    CharacterStats stats = CHARACTER_STATS.getOrDefault(type,
CHARACTER_STATS.get("Warrior"));
```

- Holt die zugehörigen Werte (z. B. Schaden, Verteidigung) aus der Map `CHARACTER_STATS`.
- Wenn kein Eintrag vorhanden ist, werden die Werte des „Warrior“ verwendet.

```
    initializeCharacter(type, x, y, stats);
}
```

- Ruft die Initialisierungsmethode auf, um Position, Bildgröße und Charakterwerte zu setzen.

```
private void initializeCharacter(String type, double x, double y,
CharacterStats stats) {
```

- Methode zur Initialisierung der Eigenschaften eines Charakters.

```
    Image characterImage = getImage();
```

- Holt das Bild, das aktuell mit `ImageView` (also mit dem Charakter) verknüpft ist.

```
    double baseHeight = 300;
```

- Legt eine einheitliche Höhe für alle Charaktere fest (Basiswert zum Skalieren).

```
    double scale = baseHeight / characterImage.getHeight();
```

- Berechnet einen Skalierungsfaktor, um das Bild proportional an `baseHeight` anzupassen.

```
    setFitHeight(characterImage.getHeight() * scale);
    setFitWidth(characterImage.getWidth() * scale);
```

- Wendet die Skalierung auf Höhe und Breite an.

```
    setPreserveRatio(true);
```

- Sorgt dafür, dass das Seitenverhältnis des Bildes erhalten bleibt.

```
    setSmooth(true);
    setCache(true);
```

- Aktiviert Bild-Glättung und Bild-Caching für bessere Performance und Darstellung.

```
    setX(x);
```

- Setzt die horizontale Startposition des Charakters.

```
    setY(GROUND_Y - getFitHeight());
```

- Platziert den Charakter vertikal auf dem Boden.
- `GROUND_Y` ist die Bodenhöhe, `getFitHeight()` die Höhe des Charakters.

Eigenschaften setzen (aus `CharacterStats`):

```
    this.type = type;
    this.damage = stats.damage;
    this.speed = stats.speed;
    this.attackRange = stats.range;
    this.defense = stats.defense;
    this.attackSpeed = stats.attackSpeed;
    this.description = stats.description;
```

- Übernimmt alle Werte aus dem `CharacterStats`-Objekt.

```
    this.maxHealth = 100;
    this.health = maxHealth;
    this.canAttack = true;
}
```

- Setzt Leben auf 100.
- Aktiviert sofortige Angriffsbereitschaft.

`jump()` – Springt, wenn nicht bereits in der Luft

```
public void jump() {
```

- Öffnet die Methode, die das Springen ausführt.

```
    if (!isJumping) {
```

- Nur springen, wenn der Charakter sich aktuell **nicht** in der Luft befindet.

```
        velocityY = JUMP_FORCE;
```

- Setzt die vertikale Geschwindigkeit auf einen negativen Wert (Springen nach oben).

```
        isJumping = true;
    }
}
```

- Markiert den Charakter als „springend“, um Doppelsprünge zu verhindern.
- Ende der Methode.

`moveLeft()` – Bewegung nach links

```
public void moveLeft() {
```

- Öffnet die Methode zur Bewegung nach links.

```
    double newX = getX() - speed;
```

- Berechnet die neue X-Position, indem die Geschwindigkeit vom aktuellen X-Wert abgezogen wird.

```
    if (newX >= 0) {
        setX(newX);
```

```
    }
```

- Stellt sicher, dass der Charakter nicht über den linken Bildschirmrand hinausläuft.
- Wenn gültig, wird die Position aktualisiert.

```
    if (facingRight) {
        setScaleX(-1);
        facingRight = false;
    }
}
```

- Dreht die Bildausrichtung nach links (negativer X-Scale), falls der Charakter vorher nach rechts schaute.
- Aktualisiert die Blickrichtung.

`moveRight()` – Bewegung nach rechts

```
public void moveRight() {
```

- Öffnet die Methode zur Bewegung nach rechts.

```
    double newX = getX() + speed;
```

- Berechnet die neue X-Position, indem die Geschwindigkeit **addiert** wird.

```
    if (newX <= 800 - getFitWidth()) {
        setX(newX);
    }
```

- Verhindert, dass der Charakter über den rechten Rand des Spielfelds (800px breit) hinausläuft.

```
    if (!facingRight) {
        setScaleX(1);
```

```
            facingRight = true;
        }
    }
```

- Falls der Charakter zuvor nach links schaute, wird er wieder nach rechts gedreht.

`update()` – Wird jedes Frame aufgerufen (Physik, Cooldowns)

```
public void update() {
```

- Öffnet die zentrale Methode, die **in jedem Frame** aufgerufen wird.

1. Sprungverhalten aktualisieren:

```
    if (isJumping) {
        setY(getY() + velocityY);
        velocityY += GRAVITY;
```

- Solange der Charakter springt:
  - Aktuelle Y-Position wird mit `velocityY` verändert.
  - Die Geschwindigkeit wird durch Schwerkraft beeinflusst (nach unten beschleunigt).

```
        if (getY() >= GROUND_Y - getFitHeight()) {
            setY(GROUND_Y - getFitHeight());
            velocityY = 0;
            isJumping = false;
        }
    }
```

- Wenn der Charakter wieder den Boden berührt:
  - Y-Position wird auf den Boden gesetzt.
  - Springen wird beendet (`isJumping = false`).

2. Angriff-Cooldown prüfen:

```
    if (!canAttack && System.currentTimeMillis() - lastAttackTime >
ATTACK_COOLDOWN) {
        canAttack = true;
        setEffect(null);
    }
}
```

- Wenn gerade **kein Angriff möglich** war, prüft die Methode, ob der Cooldown abgelaufen ist.
- Wenn ja, darf der Charakter wieder angreifen und etwaige Effekte (z. B. Leuchten) werden entfernt.

`public boolean attack(Character target, boolean isStrongAttack)`

```
public boolean attack(Character target, boolean isStrongAttack) {
```

- Führt einen Angriff auf das Ziel aus. `isStrongAttack` bestimmt, ob es sich um einen normalen oder starken Angriff handelt.

```
    if (!canAttack) return false;
```

- Wenn der Charakter gerade nicht angreifen darf (Cooldown), wird `false` zurückgegeben.

```
    int damageDealt = isStrongAttack ? damage * 2 : damage;
```

- Starker Angriff macht doppelten Schaden, sonst normalen.

```
    target.takeDamage(damageDealt);
```

- Der Schaden wird auf das Ziel angewendet.

```
    canAttack = false;
    lastAttackTime = System.currentTimeMillis();
```

- Der Charakter darf für die Dauer des Cooldowns nicht mehr angreifen.
- Der Zeitpunkt des Angriffs wird gespeichert.

```
    setEffect(new Glow(0.8));
```

- Ein visueller Effekt zeigt an, dass ein Angriff ausgeführt wurde.

```
    return true;
}
```

- Gibt `true` zurück, um anzuzeigen, dass der Angriff erfolgreich war.

`public void takeDamage(int damage)`

```
public void takeDamage(int damage) {
```

- Wird aufgerufen, wenn der Charakter getroffen wird.

```
    int actualDamage = Math.max(0, damage - defense);
```

- Der tatsächliche Schaden wird durch die Verteidigung reduziert.
- Schaden kann aber nie negativ sein.

```
    health -= actualDamage;
```

- Gesundheit wird reduziert.

```
    if (health < 0) health = 0;
}
```

- Gesundheit wird auf 0 begrenzt (kein negativer Wert).

`public void reset()`

```
public void reset() {
```

- Setzt den Zustand des Charakters für ein neues Spiel zurück.

```
    this.health = maxHealth;
    this.canAttack = true;
    this.canUseStrongAttack = true;
    this.velocityY = 0;
    this.isJumping = false;
```

- Alle zentralen Spielwerte werden zurückgesetzt: Gesundheit, Angriffsfähigkeit, Sprungstatus etc.

```
    setEffect(null);
    setY(GROUND_Y - getFitHeight());
}
```

- Effekt (Leuchten) wird entfernt.
- Charakter wird auf den Boden zurückgesetzt.

`public boolean canUseStrongAttack()`

```
public boolean canUseStrongAttack() {
```

- Gibt zurück, ob ein starker Angriff aktuell erlaubt ist.

```
    return canUseStrongAttack && System.currentTimeMillis() -
lastStrongAttackTime > STRONG_ATTACK_COOLDOWN;
}
```

- Starker Angriff darf nur ausgeführt werden, wenn:
  - `canUseStrongAttack` `true` ist **und**
  - der Cooldown seit dem letzten starken Angriff abgelaufen ist.

`public void setStrongAttackCooldown()`

```
public void setStrongAttackCooldown() {
```

- Setzt den Cooldown für den starken Angriff.

```
    this.canUseStrongAttack = false;
    this.lastStrongAttackTime = System.currentTimeMillis();
```

- Ab diesem Zeitpunkt ist ein starker Angriff nicht mehr erlaubt.

```
    new Timer().schedule(new TimerTask() {
        @Override
        public void run() {
            canUseStrongAttack = true;
        }
    }, STRONG_ATTACK_COOLDOWN);
}
```

- Ein Timer aktiviert den starken Angriff wieder nach Ablauf der Cooldown-Zeit.

Getter-Methoden

```
public int getHealth() {
    return health;
}
```

- Gibt die aktuelle Gesundheit zurück.

```
public String getType() {
    return type;
}
```

- Gibt den Typ des Charakters zurück (z. B. "Ninja").

```
public double getAttackRange() {
    return attackRange;
}
```

- Gibt die Reichweite des Charakters zurück.

```
public int getDamage() {
    return damage;
}
```

- Gibt den Basisschaden des Charakters zurück.

PlayerData File

Diese Klasse verwaltet die Siege der Spieler und speichert sie dauerhaft in einer JSON-Datei.

```
public final class PlayerData {
```

- Die Klasse ist `final` und kann nicht erweitert werden.

```
private static final Map<String, Integer> playerWins = new
ConcurrentHashMap<>();
```

- Eine threadsichere Map, die Spielernamen mit ihrer Anzahl an Siegen speichert.

```
private static final Path SAVE_PATH =
Paths.get("c:\\Users\\rmarc\\OneDrive\\Game\\game\\players.json");
```

- Der Pfad zur Datei, in der die Daten gespeichert werden.

```
static {
```

- Ein statischer Initialisierer – wird beim Laden der Klasse einmalig ausgeführt.

```
    try {
        Files.createDirectories(SAVE_PATH.getParent());
```

- Versucht, das Verzeichnis zu erstellen, in dem die Datei liegt.

```
    } catch (IOException e) {
        System.err.println("Fehler beim Erstellen des
Speicherordners: " + e.getMessage());
    }
}
```

- Gibt eine Fehlermeldung aus, wenn das Verzeichnis nicht erstellt werden konnte.

```
public static void loadData() {
```

- Diese Methode lädt gespeicherte Daten aus der Datei.

```
    if (Files.exists(SAVE_PATH)) {
```

- Nur wenn die Datei existiert, wird sie geladen.

```
        try (BufferedReader reader =
Files.newBufferedReader(SAVE_PATH)) {
```

- Öffnet einen Leser für die Datei.

```
            Map<String, Integer> loaded = new Gson().fromJson(
                reader,
                new TypeToken<Map<String, Integer>>(){}.getType()
            );
```

- Liest die JSON-Daten ein und wandelt sie in eine Map um.

```
            if (loaded != null) {
                playerWins.putAll(loaded);
            }
```

- Fügt die geladenen Werte in die Map `playerWins` ein.

```
        } catch (IOException e) {
            System.err.println("Error loading player data: " +
e.getMessage());
        }
    }
}
```

- Gibt einen Fehler aus, wenn etwas beim Lesen schiefläuft.

```
public static void saveData() {
```

- Methode zum Speichern der aktuellen Daten.

```
    try (BufferedWriter writer = Files.newBufferedWriter(SAVE_PATH))
{
        new Gson().toJson(playerWins, writer);
```

- Öffnet einen Schreiber und speichert die `playerWins`-Map als JSON.

```
    } catch (IOException e) {
        System.err.println("Error saving player data: " +
e.getMessage());
    }
}
```

- Gibt bei Problemen mit dem Schreiben eine Fehlermeldung aus.

```
public static void addPlayer(String playerName) {
```

- Fügt einen neuen Spieler zur Map hinzu (nur wenn Name gültig ist).

```
    if (playerName != null && !playerName.trim().isEmpty()) {
        playerWins.putIfAbsent(playerName, 0);
        saveData();
    }
}
```

- Fügt den Spieler mit 0 Siegen ein, wenn noch nicht vorhanden. Speichert danach.

```
public static void addWin(String playerName) {
```

- Erhöht die Sieganzahl für den angegebenen Spieler um 1.

```
    if (playerName != null && !playerName.trim().isEmpty()) {
        playerWins.put(playerName,
playerWins.getOrDefault(playerName, 0) + 1);
        saveData();
    }
}
```

- Holt den aktuellen Wert oder 0, addiert 1 und speichert die Map erneut.

```
public static int getWins(String playerName) {
    return playerWins.getOrDefault(playerName, 0);
}
```

- Gibt die aktuelle Sieganzahl zurück, oder 0, wenn der Spieler nicht existiert.

```
public static List<Map.Entry<String, Integer>> getTopPlayers(int
count) {
```

- Gibt die besten Spieler zurück – basierend auf ihrer Sieganzahl.

```
    return playerWins.entrySet().stream()
        .sorted(Map.Entry.<String,
Integer>comparingByValue().reversed())
        .limit(count)
        .toList();
}
```

- Sortiert die Map absteigend nach Siegen und gibt nur die ersten `count` Spieler zurück.

```
public static Set<String> getAllPlayerNames() {
    return new TreeSet<>(playerWins.keySet());
}
```

- Gibt alle Spielernamen alphabetisch sortiert zurück.

GameManager File

```
public class GameManager {
```

- Eine Klasse, die den gesamten Spielablauf steuert – von Menüs bis zur Charakterauswahl.

```
private Stage primaryStage;
```

- Das Hauptfenster der Anwendung.

```
private Scene mainScene;
```

- Die Szene für das Hauptmenü.

```
private Scene characterSelectScene;
```

- Die Szene für die Charakterauswahl.

```
private GameArena gameArena;
```

- Die Spielarena, in der der eigentliche Kampf stattfindet.

```
private String selectedP1Character = null;
```

- Der aktuell ausgewählte Charakter für Spieler 1.

```
private String selectedP2Character = null;
```

- Der aktuell ausgewählte Charakter für Spieler 2.

```
private String player1Name;
```

- Der Name des ersten Spielers.

```
private String player2Name;
```

- Der Name des zweiten Spielers.

```
public GameManager(Stage stage) {
    this.primaryStage = stage;
}
```

- Konstruktor, der das Hauptfenster initialisiert.

```
public VBox getMainMenu() {
```

- Methode, die das Hauptmenü erstellt und zurückgibt.

```
VBox mainMenu = new VBox(20);
mainMenu.setAlignment(Pos.CENTER);
```

- Erstellt einen vertikalen Container mit 20 Pixeln Abstand zwischen den Elementen und zentriertem Inhalt.

```
Button storyMode = new Button("Story Mode");
Button pvpMode = new Button("1v1 Mode");
Button closeButton = new Button("Exit Game");
```

- Erstellt drei Buttons: für Story-Modus, 1v1-Modus und Spiel beenden.

```
storyMode.setOnAction(e -> showStoryMode());
pvpMode.setOnAction(e -> showCharacterSelect());
closeButton.setOnAction(e -> {
    PlayerData.saveData();
```

```
    primaryStage.close();
});
```

- Definiert das Verhalten der Buttons: Story-Modus starten, Charakterauswahl anzeigen oder das Spiel beenden.

```
storyMode.setPrefWidth(150);
pvpMode.setPrefWidth(150);
closeButton.setPrefWidth(150);
```

- Setzt die Breite der Buttons auf 150 Pixel für ein einheitliches Erscheinungsbild.

```
mainMenu.getChildren().addAll(storyMode, pvpMode, closeButton);
return mainMenu;
}
```

- Fügt die Buttons zum Menü-Container hinzu und gibt diesen zurück.

```
private void showStoryMode() {
```

- Methode, die den Story-Modus anzeigt (derzeit nicht implementiert).

```
VBox storyBox = new VBox(20);
storyBox.setAlignment(Pos.CENTER);
```

- Erstellt einen vertikalen Container und zentriert den Inhalt.

```
Label comingSoon = new Label("Coming Soon!");
Button back = new Button("Back to Main Menu");
```

- Zeigt den "Coming Soon!"-Text und einen Button zur Rückkehr ins Hauptmenü.

```
back.setOnAction(e ->
primaryStage.getScene().setRoot(getMainMenu()));
```

- Beim Klick auf den Button wird das Hauptmenü wieder angezeigt.

```
storyBox.getChildren().addAll(comingSoon, back);
primaryStage.getScene().setRoot(storyBox);
}
```

- Fügt die Elemente dem Container hinzu und ersetzt die aktuelle Szene.

```
public void showCharacterSelect() {
```

- Methode, die die Charakterauswahl anzeigt.

```
PlayerData.loadData();
```

- Lädt die Spielerdaten (z. B. Siege) aus der JSON-Datei.

```
VBox characterSelect = new VBox(20);
characterSelect.setAlignment(Pos.CENTER);
```

- Erstellt einen vertikalen Container für die Auswahl und zentriert ihn.

```
Label title = new Label("Select Characters");
VBox topPlayersBox = new VBox(5);
topPlayersBox.setAlignment(Pos.CENTER);
Label topPlayersTitle = new Label("Top 3 Players");
topPlayersBox.getChildren().add(topPlayersTitle);
```

- Zeigt einen Titel und die Top-3-Spieler an.

```
List<Map.Entry<String, Integer>> topPlayers =
PlayerData.getTopPlayers(3);
for (int i = 0; i < topPlayers.size(); i++) {
    Map.Entry<String, Integer> player = topPlayers.get(i);
    Label playerLabel = new Label(String.format("%d. %s - %d wins",
        i + 1, player.getKey(), player.getValue()));
    topPlayersBox.getChildren().add(playerLabel);
}
```

- Erstellt für jeden Top-Spieler ein Label mit Rang, Name und Anzahl der Siege.

```
HBox mainContainer = new HBox(50);
mainContainer.setAlignment(Pos.CENTER);
```

- Erstellt einen horizontalen Container für die Spielerbereiche.

```
VBox p1Section = new VBox(10);
p1Section.setAlignment(Pos.CENTER);
Label p1Label = new Label("Player 1: Not Selected");
VBox p1ButtonBox = new VBox(5);
p1ButtonBox.setAlignment(Pos.CENTER);
Label statsLabelP1 = new Label("Player 1: No character selected");
statsLabelP1.setStyle("-fx-font-size: 14; -fx-font-family:
monospace;");
```

- Erstellt die UI-Elemente für Spieler 1 inkl. Charakter-Label und Statistik-Anzeige.

```
VBox p2Section = new VBox(10);
p2Section.setAlignment(Pos.CENTER);
Label p2Label = new Label("Player 2: Not Selected");
VBox p2ButtonBox = new VBox(5);
p2ButtonBox.setAlignment(Pos.CENTER);
Label statsLabelP2 = new Label("Player 2: No character selected");
statsLabelP2.setStyle("-fx-font-size: 14; -fx-font-family:
monospace;");
```

- Erstellt die UI-Elemente für Spieler 2 inkl. Charakter-Label und Statistik-Anzeige.

```
String[] characters = {
    "Bishop",
    "Holyknight",
    "Knight",
    "Magician",
    "Ninja",
    "Priestess",
    "Rogue",
    "Swordsman",
    "Warrior",
    "Wizard"
};
```

- Definiert ein Array mit allen verfügbaren Charakteren im Spiel.

```
for (String character : characters) {
```

- Durchläuft alle verfügbaren Charaktere in der Liste.

```
    Button p1Button = new Button(character);
```

- Erstellt einen Button für Spieler 1 mit dem Namen des aktuellen Charakters.

```
    Button p2Button = new Button(character);
```

- Erstellt einen Button für Spieler 2 mit dem Namen des aktuellen Charakters.

```
    p1Button.setPrefWidth(100);
```

- Setzt die bevorzugte Breite des Spieler 1-Buttons auf 100 Pixel.

```
    p2Button.setPrefWidth(100);
```

- Setzt die bevorzugte Breite des Spieler 2-Buttons auf 100 Pixel.

```
    p1Button.setOnAction(e -> {
        selectedP1Character = character;
        p1Label.setText("Player 1: " + character);
```

- Reaktion auf den Klick eines Spieler-1-Buttons: speichert Auswahl und aktualisiert das Label.

```
        Character.CharacterStats stats =
Character.CHARACTER_STATS.get(character);
        String statsText = String.format("""
            Player 1: %s
            Damage: %d
            Defense: %d
            Speed: %.1f
            Attack Speed: %d
            Attack Range: %.0f
            %s""",
            character,
            stats.damage(),
            stats.defense(),
            stats.speed(),
            stats.attackSpeed(),
            stats.range(),
            stats.description()
        );
        statsLabelP1.setText(statsText);
    });
```

- Holt die Werte des Charakters und formatiert sie für das Statistik-Label von Spieler 1.

```
    p2Button.setOnAction(e -> {
        selectedP2Character = character;
```

```
            p2Label.setText("Player 2: " + character);
```

- Reaktion auf den Klick eines Spieler-2-Buttons: speichert Auswahl und aktualisiert das Label.

```
            Character.CharacterStats stats =
Character.CHARACTER_STATS.get(character);
            String statsText = String.format("""
                Player 2: %s
                Damage: %d
                Defense: %d
                Speed: %.1f
                Attack Speed: %d
                Attack Range: %.0f
                %s""",
                character,
                stats.damage(),
                stats.defense(),
                stats.speed(),
                stats.attackSpeed(),
                stats.range(),
                stats.description()
            );
            statsLabelP2.setText(statsText);
        });
```

- Holt die Werte des Charakters und formatiert sie für das Statistik-Label von Spieler 2.

```
        p1ButtonBox.getChildren().add(p1Button);
        p2ButtonBox.getChildren().add(p2Button);
    }
```

- Fügt die Buttons zur jeweiligen Spieler-Box hinzu und schließt die Schleife.

```
p1Section.getChildren().addAll(p1Label, p1ButtonBox, statsLabelP1);
p2Section.getChildren().addAll(p2Label, p2ButtonBox, statsLabelP2);
```

- Fügt Label, Button-Gruppe und Statistik-Text zur jeweiligen Spieler-Sektion hinzu.

```
mainContainer.getChildren().addAll(p1Section, p2Section);
```

- Platziert beide Spieler-Sektionen nebeneinander im Hauptbereich.

```
Button startButton = new Button("Start Game");
startButton.setDisable(true);
```

- Erstellt einen "Start Game"-Button, der initial deaktiviert ist.

```
startButton.setOnAction(e -> {
    if (selectedP1Character != null && selectedP2Character != null) {
        getPlayerNames().ifPresent(names -> {
            player1Name = names[0];
            player2Name = names[1];
            startGame(selectedP1Character, selectedP2Character);
        });
    }
});
```

- Wenn beide Spieler einen Charakter gewählt haben, werden Namen abgefragt und das Spiel gestartet.

```
Button backButton = new Button("Back to Main Menu");
backButton.setOnAction(e -> {
    selectedP1Character = null;
    selectedP2Character = null;
    primaryStage.getScene().setRoot(getMainMenu());
});
```

- Zurück-Button: setzt Auswahl zurück und lädt das Hauptmenü.

Wenn du willst, mache ich als Nächstes weiter mit:

- `getPlayerNames()`
- `startGame(...)`
- oder alles danach im `GameManager` – sag einfach Bescheid.

```
p1Label.textProperty().addListener((obs, old, newVal) ->
    startButton.setDisable(selectedP1Character == null ||
selectedP2Character == null));
```

- Fügt einen Listener zum Text des Spieler-1-Labels hinzu, der den Start-Button aktiviert oder deaktiviert, je nachdem ob beide Spieler einen Charakter ausgewählt haben.

```
p2Label.textProperty().addListener((obs, old, newVal) ->
    startButton.setDisable(selectedP1Character == null ||
selectedP2Character == null));
```

- Fügt einen Listener zum Text des Spieler-2-Labels hinzu, der den Start-Button aktiviert oder deaktiviert, je nachdem ob beide Spieler einen Charakter ausgewählt haben.

```
HBox buttons = new HBox(20);
buttons.setAlignment(Pos.CENTER);
buttons.getChildren().addAll(backButton, startButton);
```

- Erstellt einen horizontalen Button-Container mit 20 Pixel Abstand, zentriert ihn und fügt Start- und Zurück-Button hinzu.

```
characterSelect.getChildren().addAll(topPlayersBox, title,
mainContainer, buttons);
```

- Fügt alle UI-Komponenten dem Hauptcontainer für die Charakterauswahl hinzu.

```
primaryStage.getScene().setRoot(characterSelect);
```

- Setzt die Charakterauswahl als neuen Inhalt der aktuellen Szene.

```
private Optional<String[]> getPlayerNames() {
```

- Methode zur Abfrage oder Erstellung von Spielernamen über einen Dialog.

```
VBox dialog = new VBox(10);
dialog.setAlignment(Pos.CENTER);
dialog.setPadding(new javafx.geometry.Insets(20));
```

- Erstellt und konfiguriert das Layout des Dialogs.

```
VBox p1Box = new VBox(5);
Label p1Label = new Label("Select Player 1:");
ComboBox<String> p1Select = new ComboBox<>();
p1Select.setEditable(true);
p1Select.getItems().addAll(PlayerData.getAllPlayerNames());
p1Select.setPrefWidth(200);
p1Box.getChildren().addAll(p1Label, p1Select);
```

- Erstellt UI-Elemente für Spieler 1 inklusive Dropdown zur Auswahl/Erstellung.

```
VBox p2Box = new VBox(5);
Label p2Label = new Label("Select Player 2:");
ComboBox<String> p2Select = new ComboBox<>();
p2Select.setEditable(true);
p2Select.getItems().addAll(PlayerData.getAllPlayerNames());
p2Select.setPrefWidth(200);
p2Box.getChildren().addAll(p2Label, p2Select);
```

- Dasselbe wie oben, aber für Spieler 2.

```
HBox buttons = new HBox(10);
buttons.setAlignment(Pos.CENTER);
Button confirmButton = new Button("Confirm");
Button cancelButton = new Button("Cancel");
buttons.getChildren().addAll(confirmButton, cancelButton);
```

- Fügt Bestätigen- und Abbrechen-Button hinzu und zentriert sie im Dialog.

```
dialog.getChildren().addAll(p1Box, p2Box, buttons);
```

- Fügt Spieler-Boxen und Buttons zum Dialog-Fenster hinzu.

```
Stage dialogStage = new Stage();
dialogStage.initModality(Modality.APPLICATION_MODAL);
dialogStage.setTitle("Select Players");
dialogStage.setScene(new Scene(dialog));
```

- Erstellt und konfiguriert das Dialog-Fenster mit Modalität.

```
String[] result = new String[2];
confirmButton.setOnAction(e -> {
    String p1Name = p1Select.getValue();
    String p2Name = p2Select.getValue();

    if (p1Name != null && !p1Name.trim().isEmpty() &&
        p2Name != null && !p2Name.trim().isEmpty()) {

        p1Name = p1Name.trim();
        p2Name = p2Name.trim();
        PlayerData.addPlayer(p1Name);
        PlayerData.addPlayer(p2Name);
        result[0] = p1Name;
        result[1] = p2Name;
        dialogStage.close();

    } else {
        Alert alert = new Alert(Alert.AlertType.ERROR);
        alert.setTitle("Invalid Names");
        alert.setContentText("Please enter names for both players!");
        alert.showAndWait();
    }
});
```

- Validiert die Eingaben und speichert die Spielernamen, wenn gültig – andernfalls erscheint ein Fehlerdialog.

```
cancelButton.setOnAction(e -> dialogStage.close());
dialogStage.showAndWait();
return (result[0] != null && result[1] != null) ? Optional.of(result)
: Optional.empty();
```

- Schließt das Fenster bei Abbruch oder Rückgabe der Spielerwahl, wenn sie gültig ist.

```
private void startGame(String p1Character, String p2Character) {
    gameArena = new GameArena(p1Character, p2Character, this);
    primaryStage.setScene(new Scene(gameArena.getGamePane(), 800,
600));
    gameArena.getGamePane().requestFocus();
}
```

- Startet die Spielarena mit den gewählten Charakteren und setzt sie als neue Szene.

```
public String getPlayer1Name() { return player1Name; }
public String getPlayer2Name() { return player2Name; }
```

- Getter-Methoden für die Spielernamen.

Zusatznotizen für Fragen

Eine stage is ein default ding von javafx also die primary stage is halt die default stage und mit new Stage() kann man auch neue machen.