Sprawozdanie z projektu gry Tetris w języku Java

Sposób gry

- opuszczamy szybko klocek naciskając spację, obracamy strzałkami w górę
- za każdą "zbitą" linię uzyskujemy 1 punkt

Struktura projektu:

TetrisGame/
├── .idea/
├── out/
├── src/
| └── com/
| ├── Board.java
| ├── Shape.java
| └── Tetris.java
├── .gitignore
├── TetrisGame.iml
└── External Libraries
└── Scratches and Consoles

Ustawienia wstępne

Tworzę pakiet `com`, następnie wewnątrz tworzę trzy klasy: `Board.java`, `Tetris.java` (main class) i `Shape.java` (kontroluje kształt moich obiektów

Game frame and main class

Zacznijmy od stworzenia klasy main, której głównym celem jest puszczenie całej gry. W tej klasie zaimportujemy resztę klas które będziemy używać.

```
package com;

public class Tetris {
    public static void main(String[] args) {

    }
}
```

Klasa main puści klasę `Board` z główną funkcjonalnością gry.

1. **Rozszerzamy klasę Tetris o `JFrame` - klasę która pokaże ramę okna naszej gry**

```
import javax.swing.JFrame
package com;

public class Tetris extends JFrame...
```

2. **Tworzymy okno**

- tworzymy prywatna zmienną o typie `JLabel`

```
import javax.swing.JLabel;
...
```

```
private JLabel statusbar;
```

- tworzymy konstruktor, który zaktywuje się bezpośrednio po puszczeniu. Funkcja, która ma taką samą nazwę jak nasza klasa, oznacza, że za każdym razem, gdy uruchomimy tę klasę, zostanie zainicjalizowana nasza funkcja lub nasz obiekt właśnie za pomocą tej funkcji.

```
// Konstruktor uruchamiany przy starcie programu.
// Inicjalizuje interfejs użytkownika i uruchamia planszę gry.
public Tetris() {
    initUI();
}
```

- implementujemy funkcje `initUI()`

```
private void initUI() {
    // Tworzymy pasek stanu do wyświetlania aktualnego wyniku lub informacji.
    // Bez tego użytkownik nie miałby informacji zwrotnej.
    statusbar = new JLabel("0");

    // Ustawiamy pasek stanu na dole okna.
    // Pominięcie pozycji mogłoby umieścić label w złym miejscu lub wcale.
    add(statusbar, BorderLayout.SOUTH); //pozycja statusbar

    // Tworzymy planszę gry i przekazujemy aktualne okno jako parametr (do komunikacji z paskiem stanu).
    // Bez tego `Board` nie mógłby aktualizować punktacji.
    var board = new Board(this); //this bo uzywamy aktualnej klasy
    add(board); //dodajemy plansze do okna
    board.start(); // rozpoczynamy gre automatycznie po uruchomieniu

    setTitle("Tetris"); //tytul gry
    setSize(400,800); //rozmiar okna
    //domyślna opcja zakończenia

    // Ustawiamy zachowanie przy zamknięciu okna.
    // Bez tego aplikacja mogłaby pozostać w tle po zamknięciu.setDefaultCloseOperation(EXIT_ON_CLOSE);
    setLocationRelativeTo(null);
}
```

- ustawiamy `statusbar`

```
// Getter do pobrania statusbara przez klasę Board.
// Bez niego Board nie miałby jak aktualizować informacji o punktacji.
JLabel getStatusBar() {
    return statusbar //getter dla statusbar
}
```

- piszemy klasę main - tworzymy kolejkę zdarzeń `EventQueue`

```
// Uruchamiamy grę w kolejce zdarzeń Swinga, aby GUI działało stabilnie.
// Jeśli użyjemy `new Tetris()` bez tej kolejki, mogą pojawić się problemy z wątkami i renderowaniem.
public class Tetris {
    public static void main(String[] args) {
    EventQueue.invokeLater(() -> {
        var game = new Tetris();
        game.setVisible(true);
    }); //wyrazenie Lambda () - nie przyjmuje żadnych argumentów, -> oddziela liste argumentów od ciała
funkcji
}
```

Pełna klasa `Tetris.java`

```
package com;

import java.awt.BorderLayout;
import java.awt.EventQueue;
```

```java
import javax.swing.JFrame;
import javax.swing.JLabel;


public class Tetris extends JFrame {

    private JLabel statusbar;

    public Tetris() {

        initUI();
    }


    private void initUI() {

        statusbar = new JLabel(" 0");
        add(statusbar, BorderLayout.SOUTH);

        var board = new com.Board(this);
        add(board);
        board.start();

        setTitle("Tetris");
        setSize(400, 800);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLocationRelativeTo(null);
    }

    JLabel getStatusBar() {

        return statusbar;
    }

    public static void main(String[] args) {

        EventQueue.invokeLater(() -> {

            var game = new Tetris();
            game.setVisible(true);
        });
    }
}
```

Shape.java

- zajmiemy się projektowaniem kształtów i metod które będą definiować kształt obiektów umieszczanych w grze Tetris
- aby to zrobić, użyjemy kombinacji metod i wyliczeń (enumeratorów)

Tworzymy enumerator

```java
// Enum definiujący wszystkie możliwe kształty klocków.
// NoShape reprezentuje brak figury (puste pole).
protected enum Tet { NoShape, ZShape, SShape, LineShape, TShape, SquareShape, LShape, MirroredLShape }

private Tet pieceShape; //Tworzymy tym samym zmienną typu naszego enumeratora. Enumerator działa tutaj jak szablon (blueprint) dla obiektu pieceShape
private int coords[][]; //tablica macierzy - wspolrzedne aktuanego ksztaltu
private int[][][] coordsTable; //wszystkie możliwe wspolrzedne wszytskich ksztaltow
```

Tworzymy konstruktor

```java
// Konstruktor klasy Shape – automatycznie ustawia początkowy kształt.
// Bez tego Shape byłby niezainicjalizowany i mogłoby dojść do błędów.
```

```
public Shape() {
    initShape();
}
```

Tworzymy metodę `initShape()`, która zostanie wywołana w konstruktorze.
Tam przypiszemy początkowe wartości współrzędnych i stworzymy tabelę współrzędnych:

```
coords = new int[4][2];
coordsTable = new int[8][4][2];
```

W tej tabeli zapiszemy wszystkie możliwe współrzędne każdego z 7 (lub 8 z NoShape) kształtów.

```
public void initShape() {
    // Tworzymy macierz 4x2 dla współrzędnych obecnej figury.
    coords = new int [4][2];

    // Wypełniamy tabelę współrzędnych dla każdego możliwego kształtu.
    // Pominięcie tej tabeli uniemożliwiłoby ustalanie kształtu przez `setShape`.
    coordsTable = new int [] [] [] {
    { { 0, 0 },   { 0, 0 },   { 0, 0 },   { 0, 0 } },
    { { 0, -1 },  { 0, 0 },   { -1, 0 },  { -1, 1 } },
    { { 0, -1 },  { 0, 0 },   { 1, 0 },   { 1, 1 } },
    { { 0, -1 },  { 0, 0 },   { 0, 1 },   { 0, 2 } },
    { { -1, 0 },  { 0, 0 },   { 1, 0 },   { 0, 1 } },
    { { 0, 0 },   { 1, 0 },   { 0, 1 },   { 1, 1 } },
    { { -1, -1 }, { 0, -1 },  { 0, 0 },   { 0, 1 } },
    { { 1, -1 },  { 0, -1 },  { 0, 0 },   { 0, 1 } }
    };
    setShape(Tet.NoShape); //ustawiamy domyślny kształt
}
```

- tworzymy metodę `setShape()`. W tej metodzie przypiszemy odpowiednie współrzędne dla wybranego kształtu. Dla każdego kształtu wykonamy cztery iteracje, by przypisać cztery punkty(prawo/lewo/góra/dół):

```
// Ustawiamy konkretne współrzędne kształtu na podstawie wybranego typu.
// Bez tego klocek nie miałby przypisanych współrzędnych.
protected void setShape(Tet shape) {
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 2; ++j) {
            coords[i][j] = coordsTable[shape.ordinal()][i][j];
        }
    }
}
```

- Tworzymy teraz pomocnicze metody do ustawiania i pobierania współrzędnych:

```
    public void setX(int index, int x){
    coords[index][0] = x;
    }

    private void setY(int index, int y) {
    coords[index][1] = y;
    }
    //input wartości x
    public int x(int index) {
    return coords[index][0];
    }
    //input wartości y
    public int y(int index) {
    return coords[index][1];
    }
    //pobiera Shape który potrzebujemy z enumeratora
    public Tet  getShape() {
    return pieceShape;
    }
```

- Tworzymy też metodę do ustawiania losowego kształtu przy użyciu biblioteki `random`. To oznacza: wybierz losowy

kształt z dostępnych siedmiu (pomijając NoShape) i ustaw go jako aktualny.

```
// Losuje losowy kształt spośród 7 dostępnych (bez NoShape).
// Pominięcie +1 skutkowałoby wylosowaniem NoShape, co nie powinno mieć miejsca w grze.
public void setRandomShape() {
    var r = new Random();
    int x = Math.abs(r.nextInt()) % 7 + 1; // pomijamy NoShape
    Tet[] values = Tet.values(); //tworzymy zmienna values o typie Tet, która przechowuje rodzaje naszych
kształtów
    setShape(values[x]);
}
```

- Dotarliśmy teraz do etapu, w którym chcemy znaleźć minimalne współrzędne x i y dla naszego kształtu.
- Zacznijmy od metody minX:

```
// Zwraca minimalną współrzędną X dla obecnego kształtu.
// Używane np. przy ustawianiu klocka na starcie.
public int minX() {
    int m = coords[0][0];
    for (int i = 0; i < 4; i++) {
        m = Math.min(m, coords[i][0]);
    }
    return m;
}
```

- Teraz podobnie dla współrzędnej y:

```
// Zwraca minimalną współrzędną Y – pomocne przy ustalaniu startowej pozycji.
public int minY() {
    int m = coords[0][1]; //0,1 bo mamy do czynienia ze wspolrzedna y
    for (int i = 0; i < 4; i++) {
        m = Math.min(m, coords[i][1]);
    }
    return m;
}
```

- Teraz czas na stworzenie metod obracających kształt w lewo i w prawo, które pozwolą nam obracać elementy na ekranie.
- **Rotacja w lewo:**

```
public Shape rotateLeft() {
    if (pieceShape == Tet.SquareShape) {
        return this; //nie chcemy niczego obracac
    }

    var result = new Shape();
    result.pieceShape = pieceShape;

    for (int i = 0; i < 4; i++) {
        //obraca sie w lewo dlatego y i -x
        result.setX(i, y(i));
        result.setY(i, -x(i));
    }

    return result; //obecny ksztalt
}
```

- Jeśli kształt to kwadrat (SquareShape), to nie wykonujemy rotacji.
- W przeciwnym razie tworzymy nowy obiekt Shape, kopiujemy kształt, a następnie przypisujemy nowe współrzędne z odpowiednią transformacją: `x = y`, `y = −x`.
- **Rotacja w prawo:**

```
public Shape rotateRight() {
    if (pieceShape == Tet.SquareShape) {
        return this;
    }

    var result = new Shape();
    result.pieceShape = pieceShape;

    for (int i = 0; i < 4; i++) {
        result.setX(i, -y(i));
        result.setY(i, x(i));
    }

    return result;
}
```

- Tutaj z kolei: `x = -y`, `y = x`.

Pełna klasa Shape.java

```
package com;

import java.util.Random;

public class Shape {

    protected enum Tet { NoShape, ZShape, SShape, LineShape,
        TShape, SquareShape, LShape, MirroredLShape }

    private Tet pieceShape;
    private int coords[][];
    private int[][][] coordsTable;


    public Shape() {

        initShape();
    }

    private void initShape() {

        coords = new int [4][2];

        coordsTable = new int [][][] {
                { { 0, 0 },   { 0, 0 },   { 0, 0 },   { 0, 0 } },
                { { 0, -1 },  { 0, 0 },   { -1, 0 },  { -1, 1 } },
                { { 0, -1 },  { 0, 0 },   { 1, 0 },   { 1, 1 } },
                { { 0, -1 },  { 0, 0 },   { 0, 1 },   { 0, 2 } },
                { { -1, 0 },  { 0, 0 },   { 1, 0 },   { 0, 1 } },
                { { 0, 0 },   { 1, 0 },   { 0, 1 },   { 1, 1 } },
                { { -1, -1 }, { 0, -1 },  { 0, 0 },   { 0, 1 } },
                { { 1, -1 },  { 0, -1 },  { 0, 0 },   { 0, 1 } }

        };

        setShape(Tet.NoShape);
    }

    protected void setShape(Tet shape) {

        for (int i = 0; i < 4 ; i++) {

            for (int j = 0; j < 2; ++j) {

                coords[i][j] = coordsTable[shape.ordinal()][i][j];
            }
        }
```

```
        pieceShape = shape;


    }

    private void setX(int index, int x) { coords[index][0] = x; }
    private void setY(int index, int y) { coords[index][1] = y; }
    public int x(int index) { return coords[index][0]; }
    public int y(int index) { return coords[index][1]; }
    public Tet getShape()  { return pieceShape; }


    public void setRandomShape() {

        var r = new Random();
        int x = Math.abs(r.nextInt()) %7 +1;

        Tet[] values = Tet.values();
        setShape(values[x]);
    }

    public int minX() {

        int m = coords[0][0];

        for (int i=0; i < 4; i++) {

            m = Math.min(m, coords[i][0]);
        }

        return m;
    }


    public int minY() {

        int m = coords[0][1];

        for (int i=0; i < 4; i++) {

            m = Math.min(m, coords[i][1]);
        }

        return m;
    }

    public Shape rotateLeft() {

        if(pieceShape == Tet.SquareShape) {
            return this;
        }

        var result = new Shape();
        result.pieceShape = pieceShape;

        for (int i=0;i <4; ++i) {

            result.setX(i, y(i));
            result.setY(i, -x(i));
        }

        return result;

    }

    public Shape rotateRight() {

        if(pieceShape == Tet.SquareShape) {

            return this;
        }
```

```
        var result = new Shape();
        result.pieceShape = pieceShape;

        for (int i=0;i <4; ++i) {

            result.setX(i, -y(i));
            result.setY(i, x(i));
        }

        return result;
    }
}
```

Board.java

Konfiguracja planszy

- Pierwszą rzeczą, jaką zrobię w klasie Board, będzie dodanie głównych parametrów, z których będziemy korzystać w różnych funkcjach tej klasy.
- Klasa Board oczywiście będzie rozszerzać `JPanel`, ponieważ chcemy wyświetlać grę na panelu:

```
package com;
import javax.swing.JPanel;
import javax.util.Timer;
import javax.swing.JLabel;

import com.Shape.Tet;

public class Board extends JPanel{
    private final int BOARD_WIDTH=20; // Szerokość planszy w blokach – definiuje ile klocków mieści się w
poziomie
    private final int BOARD_HEIGHT=22; // Wysokość planszy – od tego zależy ile linii użytkownik może
układać
    private final int PERIOD_INTERVAL=300; // czas w ms między kolejnymi spadkami klocka

    private Timer timer; // do kontrolowania czasu gry

    private boolean isFallingFinished = false; // Flaga informująca, czy klocek zakończył spadanie –
potrzebna do generowania nowego klocka
    private boolean isPaused = false; // czy gra jest wstrzymana

    private int numLinesRemoved = 0; // liczba usuniętych linii
    private int curX = 0;
    private int curY = 0;

    private JLabel statusBar; // Pasek statusu z klasy Tetris – do wyświetlania punktacji i komunikatów

    private Shape curPiece; //aktualnie spadajacy klocek

    private Tet[] board; // tablica reprezentująca planszę (kształty)
```

Tworzę konstruktory i metodę inicjalizującą

```
    // Konstruktor – otrzymuje referencję do klasy głównej Tetris
// Dzięki temu może modyfikować pasek statusu
    public Board(Tetris parent) {
    initBoard(parent);
    }

    // Inicjalizacja panelu gry – konfigurujemy obsługę klawiatury i status bar
    private void initBoard(Tetris parent) {

    // Ustawienie panelu jako aktywnego dla zdarzeń z klawiatury
```

```
    // Bez tego KeyListener nie zadziała
    setFocusable(true);

    // Pobranie paska statusu z klasy Tetris
    // Dzięki temu możemy na bieżąco aktualizować wynik
    statusBar = parent.getStatusBar(); // pobieramy pasek stanu z klasy Tetris

    // Rejestracja obsługi klawiatury
    // Jeśli pominiemy ten krok, gracz nie będzie mógł sterować klockami
    addKeyListener(new TAdapter());
    }
```

Dodamy metody pomocnicze określające rozmiar jednego pola na planszy w pikselach

```
    private int squareWidth() {
    return (int) getSize().getWidth() / BOARD_WIDTH;
    }

    private int squareHeight() {
    return (int) getSize().getHeight() / BOARD_HEIGHT;
    }

    // Zwraca typ klocka znajdującego się na danym polu (x, y) planszy
// Bez tego nie moglibyśmy sprawdzać kolizji ani rysować zawartości planszy
    private Tet shapeAt(int x, int y) {
    return board[(y * BOARD_WIDTH) + x];
    }
```

Uruchamianie i pauza gry

- tworzymy metode `start()`

```
    // Rozpoczyna nową grę: czyści planszę, tworzy nowy klocek i uruchamia timer
    void start() {
    curPiece = new Shape();
    board = new Tet[BOARD_WIDTH*BOARD_HEIGHT];

    clearBoard(); // Czyści całą planszę i ustawia wszystkie pola na NoShape
    newPiece(); // Generuje pierwszy klocek

    // Timer cyklicznie uruchamia GameCycle – główną pętlę gry
    timer = new Timer(PERIOD_INTERVAL, new GameCycle());
    timer.start();
    }
```

- tworzymy metode `pause()`

```
    // Przełącza stan gry między pauzą a wznowieniem
    private void pause() {

        isPaused = !isPaused;

        if (isPaused) {
            statusbar.setText("Game Paused");
        } else {
            statusbar.setText(String.valueOf(numLinesRemoved));
        }
        // Przerysowanie planszy – np. by ukryć/zatrzymać klocki
        repaint();

    }
```

Rysujemy elementy na ekranie

- Aby to zrobić, skorzystamy z funkcji `paintComponent`, która jest wbudowaną metodą służącą do rysowania w Swingu – musimy ją nadpisać:

```
import java.awt.Color;
import java.awt.Graphics;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyAdapter;
import java.awt.event.KeyEvent;

.
.
.

@Override
public void paintComponent(Graphics g) {
    // Czyści panel przed ponownym rysowaniem
    // Bez tego klocki będą się nakładać na siebie
    super.paintComponent(g);
    doDrawing(g);
}
```

- `Graphics g` – obiekt graficzny, który wykorzystamy do rysowania
- `super.paintComponent(g)` – czyści komponent przed rysowaniem
- `doDrawing(g)` – nasza własna metoda, w której będziemy rysować elementy gry
- **Tworzenie metody `doDrawing()`**

```
private void doDrawing(Graphics g) {
    var size = getSize(); // pobieramy rozmiar panelu

    //// Tutaj obliczamy, gdzie na ekranie ma zaczynać się plansza, czyli jej górna krawędź.
    int boardTop = (int) size.getHeight() - BOARD_HEIGHT * squareHeight();

    //Rysowanie planszy - pozwoli pomalowac i narysowac nasze ksztalty na roznych lokalizacjach na ekranie.
    //Iterujemy po wszystkich polach planszy.Jeśli w danym miejscu znajduje się kształt (czyli nie NoShape),
wywołujemy `drawSquare()`, by go narysować.
    for (int i=0; i < BOARD_HEIGHT; i++) {

            for (int j=0;j < BOARD_WIDTH; j++) {
                //szukamy wszystkich kombinacji x i y
                Tet shape = shapeAt(j, BOARD_HEIGHT -i -1);
                //rysujemy ksztalt
                if (shape != Tet.NoShape) {

                    drawSquare(g, j * squareWidth(),
                            boardTop + i*squareHeight(), shape);
                }
            }
        }
        //Po zakończeniu rysowania planszy, sprawdzamy czy aktualny klocek (curPiece) ma kształt
        if (curPiece.getShape() != Tet.NoShape) {

            for(int i=0; i < 4; i++) {

                int x = curX + curPiece.x(i);
                int y = curY - curPiece.y(i);

                //Tutaj rysujemy obecnie spadający klocek na planszy:
                //Przesuwamy jego współrzędne względem curX i curY.
                //Wyliczamy pozycję na ekranie i wywołujemy drawSquare().
                drawSquare(g, x*squareWidth(),
                        boardTop + (BOARD_HEIGHT - y - 1)*squareHeight(),
                        curPiece.getShape());
            }
        }

    }
```

Rysowanie kwadratów

```
    //Graphics g - obiekt graficzny do rysowania, x i y - pozycja kwadratu, shape - typ ksztaltu
    private void drawSquare(Graphics g, int x, int y, Tet shape) {

        //tworzymy tablice kolorow, kazdy kolor odpowiada innemu ksztaltowi
        Color colors[] = {new Color(0, 0, 0), new Color(204, 102, 102),
                new Color(102, 204, 102), new Color(102, 102, 204),
                new Color(204, 204, 102), new Color(204, 102, 204),
                new Color(102, 204, 204), new Color(218, 170 ,0)
        };

        var color = colors[shape.ordinal()];
            // Wypełniamy wnętrze klocka kolorem
        g.setColor(color);
        g.fillRect(x + 1, y + 1, squareWidth() - 2, squareHeight() - 2);
        // Rysujemy jaśniejsze obramowanie – efekt 3D
        g.setColor(color.brighter());
        g.drawLine(x, y + squareHeight() - 1, x, y);
        g.drawLine(x, y, x + squareWidth() - 1, y);
        // Ciemniejsze dolne i prawe obramowanie – cień
        g.setColor(color.darker());
        g.drawLine(x + 1, y + squareHeight() - 1,
                x + squareWidth() - 1, y + squareHeight() - 1);
        g.drawLine(x + squareWidth() - 1, y + squareHeight() - 1,
                x + squareWidth() - 1, y + 1);
    }
```

Ustawienie logiki gry i obrotu elementow

- `dropDown()` – całkowity spadek klocka - klocek spada w dół, aż nie można go już przesunąć,

```
    private void dropDown() {

        int newY = curY; //poczatkowa pozycja Y
        //dopoki klocek moze sie przesuwac, zmniejszamy wspolrzedna Y
        while (newY > 0) {
            //Jeśli ruch nie jest możliwy, przerywamy pętlę
            if(!tryMove(curPiece, curX, newY -1)) {

                break;
            }
            newY--;
        }
        // Klocek dotknął podłoża – zapisujemy jego pozycję i tworzymy nowy
        pieceDropped();
    }
```

- `oneLineDown()` – spadek o jedną linię

```
private void oneLineDown() {
    //Jeśli nie można przesunąć w dół – klocek spadł i zostaje zapisany na planszy
    if (!tryMove(curPiece, curX, curY - 1))
        pieceDropped();
}
```

- `clearBoard()` – wyczyszczenie planszy - usuwa wszystkie ksztalty z planszy, ustawia pola na NoShape

```
     private void clearBoard() {
         for(int i = 0; i < BOARD_HEIGHT * BOARD_WIDTH; i++) {
             //usuwamy wszystko z planszy
             board[i] = Tet.NoShape;
         }
     }
```

- `pieceDropped()` – co się dzieje po upadku klocka

```
    private void pieceDropped() {
        // Zapisujemy cztery części klocka w tablicy planszy
        for ( int i = 0; i < 4; i++) {

            int x = curX + curPiece.x(i); // współrzędna X części
            int y = curY - curPiece.y(i); // współrzędna Y części (liczona od góry)
            board[(y * BOARD_WIDTH) + x] = curPiece.getShape(); //zapisujemy ksztalt na planszy
        }
        //usuwamy pelne linie
        removeFullLines();
        //Tworzymy nowy klocek, jeśli gra nadal trwa
        if(!isFallingFinished) {
            newPiece();
        }
    }
```

- `newPiece()` – generowanie nowego klocka

```
    // Tworzy nowy losowy klocek i ustawia go na górze planszy
private void newPiece() {
    curPiece.setRandomShape(); // losujemy kształt
    curX = BOARD_WIDTH / 2 + 1; // pozycja startowa X – środek planszy
    curY = BOARD_HEIGHT - 1 + curPiece.minY(); // pozycja Y – najwyżej jak się da

    // Jeśli nie można umieścić nowego klocka – koniec gry
    if (!tryMove(curPiece, curX, curY)) {
        curPiece.setShape(Tet.NoShape); // klocek „znika”
        timer.stop(); // zatrzymujemy grę

        // Komunikat „Game Over” z wynikiem
        var msg = String.format("Koniec gry! Wynik: %d", numLinesRemoved);
        statusBar.setText(msg); // wyświetlamy wynik w pasku stanu
    }
}
```

Ruszanie kształtami

- metoda `tryMove()` – próba przesunięcia klocka

```
// Próbuje przesunąć klocka na nowe współrzędne (x, y)
// Zwraca true – jeśli przesunięcie jest możliwe, lub false – jeśli ruch jest nielegalny
private boolean tryMove(Shape newPiece, int newX, int newY) {
    // Sprawdzamy wszystkie 4 części klocka
    for (int i = 0; i < 4; i++) {
        int x = newX + newPiece.x(i); // nowe X dla i-tego bloku
        int y = newY - newPiece.y(i); // nowe Y dla i-tego bloku

        // Sprawdzenie, czy pozycja wychodzi poza planszę
        if (x < 0 || x >= BOARD_WIDTH || y < 0 || y >= BOARD_HEIGHT) {
            return false;
        }

        // Sprawdzenie, czy w nowej pozycji coś już się znajduje
        if (shapeAt(x, y) != Tet.NoShape) {
            return false;
        }
    }

    // Jeśli wszystko OK, aktualizujemy pozycję i klocek
    curPiece = newPiece;
    curX = newX;
    curY = newY;

    repaint(); // odśwież planszę
    return true; //bo boolean
}
```

Usuwanie pełnych linii

- metoda `removeFullLines()` – usuwanie pełnych linii
- W `removeFullLines()` robimy tzw. przesuwanie wierszy w dół.
- metoda działa "od dołu do góry", ponieważ linie zapełniane są od dołu.
- Po każdorazowym usunięciu linii sprawdzamy ponownie ten sam wiersz (bo wyżej coś mogło spaść).

```
// Usuwa wszystkie w pełni zapełnione linie z planszy
private void removeFullLines() {
    int numFullLines = 0; // licznik usuniętych linii

    // Sprawdzamy każdą linię od dołu do góry
    for (int i = BOARD_HEIGHT - 1; i >= 0; i--) {
        boolean lineIsFull = true;

        // Sprawdzenie, czy dana linia jest pełna
        for (int j = 0; j < BOARD_WIDTH; j++) {
            if (shapeAt(j, i) == Tet.NoShape) {
                lineIsFull = false;
                break; // jeśli jest puste pole – przerywamy pętlę
            }
        }

        // Jeśli linia pełna – przesuwamy wszystko w górę
        if (lineIsFull) {
            numFullLines++; // zwiększamy licznik usuniętych linii

            // Przesuwamy linie z góry w dół (nadpisujemy bieżącą)
            for (int k = i; k < BOARD_HEIGHT - 1; k++) {
                for (int j = 0; j < BOARD_WIDTH; j++) {
                    board[k * BOARD_WIDTH + j] = shapeAt(j, k + 1);
                }
            }

            // Po przesunięciu, sprawdzamy tę samą linię jeszcze raz (bo z góry coś spadło)
            i++;
        }
    }

    // Jeśli jakieś linie zostały usunięte
    if (numFullLines > 0) {
        numLinesRemoved += numFullLines; // aktualizujemy wynik
        statusBar.setText(String.valueOf(numLinesRemoved)); // pokazujemy na pasku stanu
        isFallingFinished = true; // zakończono spadanie klocka
        curPiece.setShape(Tet.NoShape); // usuwamy aktualny klocek
    }
}
```

Cykl gry

- Cykl gry (GameCycle) – logika wykonywana cyklicznie

```
// Klasa wewnętrzna obsługująca cykliczne wykonywanie gry
private class GameCycle implements ActionListener {

    // Metoda wywoływana co określony czas (300 ms)
    @Override
    public void actionPerformed(ActionEvent e) {
        doGameCycle(); // wywołujemy logikę gry
    }
}
// Główna funkcja cyklu gry
private void doGameCycle() {
    update();   // aktualizacja logiki gry (np. spadek klocka)
    repaint();  // przerysowanie planszy
}
// Funkcja aktualizująca grę
private void update() {
    if (isPaused) return; // jeśli gra zatrzymana – nic nie rób
```

```
    if (isFallingFinished) {
        isFallingFinished = false;
        newPiece(); // rozpocznij nowy klocek
    } else {
        oneLineDown(); // przesuń aktualny klocek o 1 w dół
    }
}
```

Obsługa klawiatury (KeyAdapter)

```
// Obsługa przycisków z klawiatury
private class TAdapter extends KeyAdapter {

    @Override
    public void keyPressed(KeyEvent e) {
        if (curPiece.getShape() == Tet.NoShape) return;

        int keycode = e.getKeyCode(); // pobieramy naciśnięty klawisz

        switch (keycode) {
            case KeyEvent.VK_P -> pause(); // pauza po 'P'

            case KeyEvent.VK_LEFT -> // ruch w lewo
                tryMove(curPiece, curX - 1, curY);

            case KeyEvent.VK_RIGHT -> // ruch w prawo
                tryMove(curPiece, curX + 1, curY);

            case KeyEvent.VK_UP -> // obrót w lewo
                tryMove(curPiece.rotateLeft(), curX, curY);

            case KeyEvent.VK_DOWN -> // obrót w prawo
                tryMove(curPiece.rotateRight(), curX, curY);

            case KeyEvent.VK_SPACE -> // szybki spadek na dół
                dropDown();

            case KeyEvent.VK_D -> // ręczny ruch o 1 w dół
                oneLineDown();
        }
    }
}
```

Aktywacja obsługi klawiatury i timera

Dodajemy do konstruktora Board:
`addKeyListener(new TAdapter()); // włączenie klawiszy`

Uruchamiamy timer w funkcji startującej grę:
`timer = new Timer(PERIOD_INTERVAL, new GameCycle()); timer.start();`

`PERIOD_INTERVAL = 300` – oznacza, że gra będzie aktualizowana co 300 ms (klocek spada).

Dodanie mnożnika punktów

```
// Dodano nowe zmienne
private int score = 0;
private int multiplier = 1;

// Zmodyfikowana metoda removeFullLines()
private void removeFullLines() {
    // ... istniejacy kod ...
    if(numFullLines > 0) {
        // Uaktualnij multiplier na podstawie ilości zbitych linii
        multiplier = numFullLines;
```

```
        // Uaktualnij wynik (n^2)
        score += numFullLines * numFullLines;
        numLinesRemoved += numFullLines;

        statusbar.setText(String.format("Score: %d | Lines: %d | Multiplier: x%d", score, numLinesRemoved,
multiplier));
        isFallingFinished = true;
        curPiece.setShape(Tet.NoShape);
    }
}

// Zmodyfikowana metoda newPiece()
private void newPiece() {
    // ... istniejacy kod ...
    if (!tryMove(curPiece, curX, curY)) {
        curPiece.setShape(Tet.NoShape);
        timer.stop();

        var msg = String.format("Game over. Final Score: %d | Lines: %d", score, numLinesRemoved);
        statusbar.setText(msg);
    }
}
```

Pełna klasa `Board.java`

```
package com;

import java.awt.Color;
import java.awt.Graphics;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyAdapter;
import java.awt.event.KeyEvent;

import javax.swing.Timer;

import javax.swing.JLabel;
import javax.swing.JPanel;

import com.Shape.Tet;

public class Board extends JPanel {

    private final int BOARD_WIDTH=10;
    private final int BOARD_HEIGHT=22;
    private final int PERIOD_INTERVAL=300;

    private Timer timer;
    private boolean isFallingFinished = false;
    private boolean isPaused =false;
    private int numLinesRemoved = 0;
    private int score = 0;
    private int multiplier = 1;
    private int curX =0;
    private int curY =0;
    private JLabel statusbar;
    private Shape curPiece;
    private Tet[] board;

    public Board(Tetris parent) {

        initBoard(parent);
    }

    private void initBoard(Tetris parent) {

        setFocusable(true);
        statusbar = parent.getStatusBar();
```

```
        addKeyListener(new TAdapter());

    }

    private int squareWidth() {

        return (int) getSize().getWidth() / BOARD_WIDTH;
    }

    private int squareHeight() {
        return (int) getSize().getHeight() / BOARD_HEIGHT;
    }

    private Tet shapeAt(int x, int y) {

        return board[(y*BOARD_WIDTH) + x];
    }

    void start() {

        curPiece = new Shape();
        board = new Tet[BOARD_WIDTH*BOARD_HEIGHT];

        clearBoard();
        newPiece();

        timer = new Timer(PERIOD_INTERVAL, new GameCycle());
        timer.start();
    }

    private void pause() {

        isPaused = !isPaused;

        if (isPaused) {

            statusbar.setText("Game Paused");
        } else {
            statusbar.setText(String.valueOf(numLinesRemoved));
        }

        repaint();

    }


    @Override
    public void paintComponent(Graphics g) {

        super.paintComponent(g);
        doDrawing(g);
    }

    private void doDrawing(Graphics g) {

        var size = getSize();
        int boardTop = (int) size.getHeight() - BOARD_HEIGHT*squareHeight();


        for (int i=0; i < BOARD_HEIGHT; i++) {

            for (int j=0;j < BOARD_WIDTH; j++) {

                Tet shape = shapeAt(j, BOARD_HEIGHT -i -1);

                if (shape != Tet.NoShape) {

                    drawSquare(g, j * squareWidth(),
                            boardTop + i*squareHeight(), shape);
                }
            }
```

```
        }

        if (curPiece.getShape() != Tet.NoShape) {

            for(int i=0; i < 4; i++) {

                int x = curX + curPiece.x(i);
                int y = curY - curPiece.y(i);

                drawSquare(g, x*squareWidth(),
                        boardTop + (BOARD_HEIGHT - y - 1)*squareHeight(),
                        curPiece.getShape());
            }
        }

    }



    private void dropDown() {

        int newY = curY;

        while (newY > 0) {

            if(!tryMove(curPiece, curX, newY -1)) {

                break;
            }
            newY--;
        }

        pieceDropped();
    }

    private void oneLineDown() {

        if(!tryMove(curPiece, curX, curY -1)) {
            pieceDropped();
        }
    }

    private void clearBoard() {
        for(int i = 0; i < BOARD_HEIGHT * BOARD_WIDTH; i++) {

            board[i] = Tet.NoShape;
        }
    }

    private void pieceDropped() {

        for ( int i = 0; i < 4; i++) {

            int x = curX + curPiece.x(i);
            int y = curY - curPiece.y(i);
            board[(y * BOARD_WIDTH) + x] = curPiece.getShape();
        }

        removeFullLines();

        if(!isFallingFinished) {
            newPiece();
        }

    }

    private void newPiece() {

        curPiece.setRandomShape();
        curX = BOARD_WIDTH / 2 + 1;
```

```
        curY = BOARD_HEIGHT - 1 + curPiece.minY();

        if (!tryMove(curPiece, curX, curY)) {

            curPiece.setShape(Tet.NoShape);
            timer.stop();

            var msg = String.format("Game over. Final Score: %d | Lines: %d", score, numLinesRemoved);
            statusbar.setText(msg);
        }
    }


    private boolean tryMove(Shape newPiece, int newX, int newY) {

        for(int i =0; i<4; i++) {

            int x = newX + newPiece.x(i);
            int y = newY - newPiece.y(i);

            if(x< 0 || x >= BOARD_WIDTH || y < 0 || y >= BOARD_HEIGHT) {

                return false;
            }

            if (shapeAt(x, y) != Tet.NoShape) {

                return false;
            }
        }

        curPiece = newPiece;
        curX = newX;
        curY = newY;

        repaint();

        return true;
    }


    private void removeFullLines() {

        int numFullLines = 0;

        for (int i = BOARD_HEIGHT -1; i >=0; i--) {

            boolean lineIsFull = true;

            for (int j = 0; j < BOARD_WIDTH; j++) {

                if(shapeAt(j,i) == Tet.NoShape) {

                    lineIsFull = false;
                    break;
                }
            }

            if (lineIsFull) {

                numFullLines++;

                for (int k = i; k < BOARD_HEIGHT - 1; k++) {
                    for (int j = 0; j < BOARD_WIDTH; j++) {

                        board[(k*BOARD_WIDTH) +j] = shapeAt(j, k +1);
                    }
                }
            }
        }
```

```
        if(numFullLines > 0) {
            // Update multiplier based on number of lines cleared
            multiplier = numFullLines;

            // Update score with squared points (n²)
            score += numFullLines * numFullLines;
            numLinesRemoved += numFullLines;

            statusbar.setText(String.format("Score: %d | Lines: %d | Multiplier: x%d", score,
numLinesRemoved, multiplier));
            isFallingFinished = true;
            curPiece.setShape(Tet.NoShape);
        }
    }

    private void drawSquare(Graphics g, int x, int y, Tet shape) {

        Color colors[] = {new Color(0, 0, 0), new Color(204, 102, 102),
                new Color(102, 204, 102), new Color(102, 102, 204),
                new Color(204, 204, 102), new Color(204, 102, 204),
                new Color(102, 204, 204), new Color(218, 170 ,0)
        };


        var color = colors[shape.ordinal()];

        g.setColor(color);
        g.fillRect(x + 1, y + 1, squareWidth() - 2, squareHeight() - 2);

        g.setColor(color.brighter());
        g.drawLine(x, y + squareHeight() - 1, x, y);
        g.drawLine(x, y, x + squareWidth() - 1, y);

        g.setColor(color.darker());
        g.drawLine(x + 1, y + squareHeight() - 1,
                x + squareWidth() - 1, y + squareHeight() - 1);
        g.drawLine(x + squareWidth() - 1, y + squareHeight() - 1,
                x + squareWidth() - 1, y + 1);
    }



    private class GameCycle implements ActionListener {

        @Override
        public void actionPerformed(ActionEvent e) {

            doGameCycle();
        }

    }

    private void doGameCycle() {

        update();
        repaint();
    }


    private void update() {

        if(isPaused) {

            return;
        }

        if (isFallingFinished) {

            isFallingFinished = false;
            newPiece();
        } else {
```

```
            oneLineDown();
        }
    }


    class TAdapter extends KeyAdapter {

        @Override
        public void keyPressed(KeyEvent e) {

            if(curPiece.getShape() == Tet.NoShape) {

                return;
            }

            int keycode = e.getKeyCode();

            switch (keycode) {

                case KeyEvent.VK_P -> pause();
                case KeyEvent.VK_LEFT -> tryMove(curPiece, curX - 1, curY);
                case KeyEvent.VK_RIGHT ->tryMove(curPiece, curX + 1, curY);
                case KeyEvent.VK_UP -> tryMove(curPiece.rotateLeft(), curX, curY);
                case KeyEvent.VK_DOWN -> tryMove(curPiece.rotateRight(), curX, curY);
                case KeyEvent.VK_SPACE -> dropDown();
                case KeyEvent.VK_D -> oneLineDown();


            }

        }
    }


}
```