# Case Study: Interactive Game in Java

## Integration of OOP, GUI, 2D Graphics and Multithreading

Zbigniew Dendzik

March 2026

# Lecture Plan

- Project overview – Breakout-style game
- Application architecture
- Object-oriented programming – classes and relationships
- Inheritance in practice
- Polymorphism and interfaces
- Java2D library classes
- Multithreading – animation
- Interactivity – mouse event handling
- Collision detection
- Optimizations and extensions

# Project Goal

**Breakout/Arkanoid-style game**

**Game elements:**

- Ball bouncing off walls
- Paddle controlled by mouse
- Bricks to break
- Scoring system
- Multi-level gameplay

**Demonstrated concepts:**

- Object-oriented programming
- Inheritance and polymorphism
- 2D graphics
- Multithreading
- Event handling

# Application Architecture

**Main system components:**

1. **Program** – main class with main()
2. **Board** – graphics panel (JPanel)
3. **Ball** – object moving around board
4. **BallEngine** – animation thread
5. **Paddle** – mouse-controlled object
6. **Brick** – breakable elements (extension)

**Class relationships:**

- Inheritance: Ball extends Ellipse2D.Float
- Inheritance: Paddle extends Rectangle2D.Float
- Implementation: Board implements MouseMotionListener
- Inheritance: BallEngine extends Thread

# Class Diagram – Relationships

**Inheritance hierarchy:**

`Ellipse2D.Float` (Java2D)
↑ extends
`Ball`

`Rectangle2D.Float` (Java2D)
↑ extends
`Paddle`, `Brick`

`Thread` (java.lang)
↑ extends
`BallEngine`

`JPanel` (Swing) + `MouseMotionListener`
↑ extends + implements
`Board`

# Object-Oriented Programming – Overview

**OOP principles in project:**

**Encapsulation:**

- Private fields (x, y, dx, dy in Ball)
- Access through methods (setX() in Paddle)

**Inheritance:**

- Ball inherits from Ellipse2D.Float
- Gains fields: x, y, width, height
- Gains methods: getMinX(), getMaxX(), intersects()

**Polymorphism:**

- Ball and Paddle are Shape objects
- Can be drawn via g2d.fill(Shape)
- MouseMotionListener – interface implementation

# Ball Class – Part 1

```
import java.awt.geom.*;

class Ball extends Ellipse2D.Float
{
    Board b;
    int dx, dy;

    Ball(Board b, int x, int y, int dx, int dy)
    {
        this.x = x;
        this.y = y;
        this.width = 10;
        this.height = 10;

        this.b = b;
        this.dx = dx;
        this.dy = dy;
    }
```

# Ball Class – Part 2

```
    void nextStep()
    {
        x += dx;
        y += dy;

        if(getMinX() < 0 || getMaxX() > b.getWidth())
            dx = -dx;

        if(getMinY() < 0 || getMaxY() > b.getHeight())
            dy = -dy;

        b.repaint();
        Toolkit.getDefaultToolkit().sync();
    }
}
```

# Ball – OOP Analysis

**Inheritance from Ellipse2D.Float:**

**Inherited fields:**

- `float x, y` – position
- `float width, height` – dimensions

**Inherited methods:**

- `getMinX(), getMaxX()` – horizontal bounds
- `getMinY(), getMaxY()` – vertical bounds
- `intersects(Rectangle2D)` – collision detection
- `contains(Point2D)` – point test

**Own fields:**

- `Board b` – reference to board
- `int dx, dy` – velocity vector

**Own methods:**

- `nextStep()` – position update and bouncing

# Ball – Movement Logic

**nextStep() method – animation step:**

1. **Position update:**
   - `x += dx;` – horizontal movement
   - `y += dy;` – vertical movement
2. **Wall collision detection:**
   - Left/right: `getMinX() < 0 || getMaxX() > width`
   - Top/bottom: `getMinY() < 0 || getMaxY() > height`
3. **Bouncing – direction change:**
   - Horizontal bounce: `dx = -dx`
   - Vertical bounce: `dy = -dy`
4. **Display refresh:**
   - `b.repaint()` – calls paintComponent()
   - `Toolkit.sync()` – synchronizes buffer

# BallEngine Class – Thread

```
class BallEngine extends Thread
{
    Ball a;

    BallEngine(Ball a)
    {
        this.a = a;
        start();
    }

    public void run()
    {
        try
        {
            while(true)
            {
                a.nextStep();
                sleep(15);
```

# BallEngine – Multithreading

**Inheritance from Thread:**

**Constructor:**

- Stores reference to Ball
- Calls `start()` – launches thread

**run() method:**

- Called automatically by JVM
- Infinite animation loop
- `a.nextStep()` – state update
- `sleep(15)` – 15ms pause ( 67 FPS)

**Exception handling:**

- `InterruptedException` – thread interruption
- Empty catch block – silent handling

# Multithreading – Concepts

**Why separate thread?**

**Event Dispatch Thread (EDT):**

- Main Swing thread
- Handles GUI events
- Calls paintComponent()
- Cannot be blocked by long operations

**Animation thread:**

- Runs parallel with EDT
- Updates game state (object positions)
- Calls repaint() – schedules redraw
- EDT performs actual drawing

**Synchronization:**

- `repaint()` – safe call from another thread
- `Toolkit.sync()` – synchronizes graphics buffer

# Paddle Class

```
class Paddle extends Rectangle2D.Float
{
    Paddle(int x)
    {
        this.x = x;
        this.y = 170;
        this.width = 60;
        this.height = 10;
    }

    void setX(int x)
    {
        this.x = x;
    }
}
```

# Paddle – Analysis

**Inheritance from Rectangle2D.Float:**

**Inherited fields:**

- `float x, y` – position
- `float width, height` – dimensions (60×10)

**Constructor:**

- Sets x position (passed as parameter)
- Fixed y position = 170 (bottom of screen)
- Fixed dimensions: 60×10 pixels

**setX() method:**

- Updates horizontal position
- Called by mouse event handler
- Simple – no validation (can be added)

# Board Class – Part 1

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

class Board extends JPanel
           implements MouseMotionListener
{
    Paddle p;
    Ball a;
    BallEngine s;

    Board()
    {
        super();
        addMouseMotionListener(this);

        p = new Paddle(100);
        a = new Ball(this, 100, 100, 1, 1);
```

# Board Class – Part 2

```
    public void paintComponent(Graphics g)
    {
        super.paintComponent(g);
        Graphics2D g2d = (Graphics2D)g;

        g2d.fill(a);
        g2d.fill(p);
    }

    public void mouseMoved(MouseEvent e)
    {
        p.setX(e.getX() - 50);
        repaint();
    }

    public void mouseDragged(MouseEvent e)
    {
    }
```

# Board – OOP Analysis

**Inheritance from JPanel:**

- Swing component for drawing
- Override `paintComponent()`

**MouseMotionListener implementation:**

- Interface with two methods
- `mouseMoved()` – mouse movement
- `mouseDragged()` – movement with pressed button

**Composition:**

- Board contains (HAS-A) Paddle, Ball, BallEngine
- Game object aggregation
- Lifecycle management

# Board – Initialization

**Constructor – configuration:**

1. **Superclass constructor call:**
   - `super()` – JPanel initialization
2. **Listener registration:**
   - `addMouseMotionListener(this)`
   - `this` – class itself implements interface
3. **Game object creation:**
   - `p = new Paddle(100)` – paddle at x=100
   - `a = new Ball(this, 100, 100, 1, 1)` – ball with velocity (1,1)
   - `s = new BallEngine(a)` – start animation thread

**Order matters:**

- Objects must exist before thread starts

# Board – Rendering

**paintComponent() method:**

**Superclass call:**

- `super.paintComponent(g)`
- Clears background
- Draws Swing components

**Context casting:**

- `Graphics2D g2d = (Graphics2D)g`
- Access to advanced Java2D methods

**Drawing objects:**

- `g2d.fill(a)` – filled ellipse (ball)
- `g2d.fill(p)` – filled rectangle (paddle)
- Polymorphism: Shape → Ellipse2D.Float, Rectangle2D.Float

# Interactivity – Mouse Handling

**MouseMotionListener interface:**

**mouseMoved() method:**

- Called on mouse movement (no button pressed)
- Parameter: `MouseEvent e`
- `e.getX()` – cursor X position
- `e.getY()` – cursor Y position

**Paddle control logic:**

- `p.setX(e.getX() - 50)`
- 50 pixel offset – centers paddle
- Paddle width 60px, so 30px left, 30px right

**Refresh:**

- `repaint()` – schedules redraw
- Immediate response to mouse movement

# Program Class – main()

```
public class Program
{
    public static void main(String[] args)
    {
        javax.swing.SwingUtilities.invokeLater(
            new Runnable()
            {
                public void run()
                {
                    Board b = new Board();

                    JFrame jf = new JFrame();
                    jf.add(b);

                    jf.setTitle("Graphics Test");
                    jf.setSize(400, 370);
                    jf.setDefaultCloseOperation(
                        JFrame.EXIT_ON_CLOSE);
```

# Program – Application Launch

**SwingUtilities.invokeLater():**

**Why invokeLater?**

- Swing is not thread-safe
- GUI must be created in EDT
- `invokeLater()` – schedules execution in EDT

**Anonymous inner class:**

- `new Runnable() { ... }`
- Runnable interface implementation
- `run()` method – code executed in EDT

**Window creation:**

- Create board
- Create JFrame
- Add board to frame
- Configure and display

# JFrame Configuration

**Basic methods:**

**setTitle():**

- Sets window title
- Visible on title bar

**setSize():**

- Sets window dimensions (width $\times$ height)
- 400$\times$370 pixels

**setDefaultCloseOperation():**

- `EXIT_ON_CLOSE` – closing terminates program
- Alternatives: DISPOSE_ON_CLOSE, HIDE_ON_CLOSE

**setVisible():**

- `true` – shows window
- Must be called at end (after configuration)

# Polymorphism in Practice

**Polymorphism usage:**

1. **Shape interface:**
   - Ball (Ellipse2D.Float) implements Shape
   - Paddle (Rectangle2D.Float) implements Shape
   - `fill(Shape)` method accepts both
2. **MouseMotionListener interface:**
   - Board implements interface
   - Can be passed to `addMouseMotionListener()`
   - Method calls through interface
3. **Thread class:**
   - BallEngine inherits from Thread
   - Can be started by `start()`
   - `run()` method called automatically

# Collision Detection – Theory

**Shape class methods:**

**intersects(Rectangle2D):**

- Checks if shapes intersect
- Accepts rectangle (bounding box)
- Returns boolean

**contains(Point2D):**

- Checks if point lies within shape
- Used for precise tests

**getBounds2D():**

- Returns bounding rectangle
- Used for fast collision tests

# Ball-Paddle Collision – Implementation

```
void nextStep()
{
    x += dx;
    y += dy;

    // Wall collisions
    if(getMinX() < 0 || getMaxX() > b.getWidth())
        dx = -dx;
    if(getMinY() < 0 || getMaxY() > b.getHeight())
        dy = -dy;

    // Paddle collision
    if(this.intersects(b.p.getBounds2D()))
    {
        dy = -dy;  // Vertical bounce
        y = b.p.y - height;  // Position correction
    }
```

# Paddle Collision – Details

**Collision detection:**

**Intersection test:**

- `this.intersects(b.p.getBounds2D())`
- `this` – Ball object (ellipse)
- `b.p` – Paddle object (rectangle)

**Collision response:**

- `dy = -dy` – reverse vertical direction
- Ball bounces upward

**Position correction:**

- `y = b.p.y - height`
- Places ball exactly above paddle
- Prevents multiple collision detections
- Eliminates ball "sinking" into paddle

# Brick Class

```
class Brick extends Rectangle2D.Float
{
    boolean active;

    Brick(int x, int y)
    {
        this.x = x;
        this.y = y;
        this.width = 40;
        this.height = 15;
        this.active = true;
    }

    boolean isActive()
    {
        return active;
    }
```

# Brick – Analysis

**Extension of Rectangle2D.Float:**

**Own field:**

- `boolean active` – is brick active
- Unbroken bricks: `true`
- Broken bricks: `false`

**Constructor:**

- Sets position (x, y)
- Fixed dimensions $40 \times 15$ pixels
- Initializes as active

**Methods:**

- `isActive()` – state getter
- `hit()` – marks brick as broken

# Brick Management in Board

```
class Board extends JPanel
           implements MouseMotionListener
{
    Paddle p;
    Ball a;
    BallEngine s;
    ArrayList<Brick> bricks;

    Board()
    {
        super();
        addMouseMotionListener(this);

        bricks = new ArrayList<Brick>();

        // Create brick grid
        for(int i = 0; i < 8; i++)
            for(int j = 0; j < 4; j++)
```

# Drawing Bricks

```
public void paintComponent(Graphics g)
{
    super.paintComponent(g);
    Graphics2D g2d = (Graphics2D)g;

    g2d.fill(a);
    g2d.fill(p);

    // Draw active bricks
    for(Brick c : bricks)
    {
        if(c.isActive())
        {
            g2d.fill(c);
        }
    }
}
```

# ArrayList – Dynamic Collection

**Why ArrayList?**

**Advantages over array:**

- Dynamic size
- Easy element addition
- Convenient iteration (for-each)
- Helper methods (size(), clear(), remove())

**Usage in project:**

- `ArrayList<Brick> bricks`
- Generic type – type safety
- `add()` – adding bricks
- For-each – iteration for drawing and collisions

**Alternative:**

- Array: `Brick[] bricks`
- Faster but less flexible

# Brick Collisions

```
void nextStep()
{
    x += dx;
    y += dy;

    // Wall and paddle collisions...

    // Brick collisions
    for(Brick c : b.bricks)
    {
        if(c.isActive() &&
           this.intersects(c.getBounds2D()))
        {
            c.hit();
            dy = -dy;
            break;  // One collision per step
        }
    }
```

# Collision Detection Optimization

**Performance problem:**

**Naive approach:**

- Testing all bricks every step
- Complexity: O(n) where n = number of bricks
- For 32 bricks: 32 tests $\times$ 67 FPS = 2144 tests/s

**Optimizations:**

- **break** – stop after first collision
- Test `isActive()` before `intersects()`
- Spatial partitioning – divide space
- Bounding box – fast preliminary test

**Advanced:**

- Quad-tree for many objects
- Remove broken bricks from list

# Scoring System

```
class Board extends JPanel
            implements MouseMotionListener
{
    // ...
    int score;

    Board()
    {
        // ...
        score = 0;
    }

    void addPoints(int points)
    {
        score += points;
    }

    public void paintComponent(Graphics g)
```

# Scoring for Breaking Bricks

```
void nextStep()
{
    x += dx;
    y += dy;

    // ... wall and paddle collisions ...

    // Brick collisions
    for(Brick c : b.bricks)
    {
        if(c.isActive() &&
           this.intersects(c.getBounds2D()))
        {
            c.hit();
            b.addPoints(10);  // 10 points per brick
            dy = -dy;
            break;
        }
```

# Advanced Paddle Bouncing

**Bounce physics:**

**Simple bounce:**

- `dy = -dy` – always same angle
- Monotonous, predictable

**Position-dependent bounce:**

- Paddle center $\rightarrow$ vertical bounce
- Paddle edges $\rightarrow$ sharp angle bounce
- Increases player control

**Implementation:**

- Calculate distance from paddle center
- Normalize to [-1, 1]
- Multiply dx by coefficient
- Preserve absolute speed (kinetic energy)

# Advanced Bounce – Code

```
if(this.intersects(b.p.getBounds2D()))
{
    // Calculate bounce point relative to paddle center
    float pCenter = b.p.x + b.p.width / 2;
    float ballCenter = x + width / 2;
    float offset = (ballCenter - pCenter) / (b.p.width / 2);

    // offset in range [-1, 1]
    // -1 = left edge, 0 = center, 1 = right edge

    float speed = (float)Math.sqrt(dx*dx + dy*dy);
    dx = offset * speed;
    dy = -(float)Math.sqrt(speed*speed - dx*dx);

    y = b.p.y - height;
}
```

# Multi-Level Game

**Level structure:**

**Advancing to next level:**

- Detect all bricks broken
- Stop animation thread
- Load new brick layout
- Reset ball and paddle positions
- Restart animation

**Storing layouts:**

- 2D array: `int[][] level`
- 0 = empty space, 1 = brick
- Different colors for different values

**Increasing difficulty:**

- Higher ball speed
- More bricks
- Narrower paddle
- Obstacles

# Visual and Audio Effects

**Graphics improvements:**

**Colors:**

- `g2d.setColor(Color)` before `fill()`
- Different colors for different brick types
- Gradients for 3D effect

**Images:**

- `BufferedImage` for textures
- `ImageIO.read()` – loading
- `TexturePaint` – texture painting

**Sounds:**

- `AudioClip` (javax.sound.sampled)
- Play on collisions
- Background music

# Bonuses and Power-ups

**Bonus types:**

**Positive:**

- Wide paddle
- Slower ball
- Extra life
- Multi-ball

**Negative:**

- Narrow paddle
- Faster ball
- Reversed controls

**Implementation:**

- Class `Bonus extends Rectangle2D.Float`
- Fall after breaking special brick
- Collision with paddle activates effect
- Timer for limited duration

# Cellular Automaton – Traffic Simulation

**Nagel-Schreckenberg model:**

**Automaton rules:**

- Road = sequence of cells
- Vehicle occupies 1 cell
- Speed: 0-5 cells/step

**Update algorithm (for each vehicle):**

1. Acceleration: `if(v < 5) v++`
2. Safety: `if(v > d-1) v = d-1`
3. Random braking: `if(random() < p) v--`
4. Movement: `x += v`

**Observations:**

- Traffic jams emerge spontaneously
- Propagate backward
- Persist long

# Traffic Simulation – Implementation

**Classes:**

**Vehicle:**

- Fields: `int position, velocity`
- Method: `update(Vehicle ahead, double p)`

**Road:**

- `ArrayList<Vehicle> vehicles`
- Method: `step()` – update all

**Visualization:**

- JPanel with road drawing
- Vehicles as rectangles
- Color based on speed
- Animation in separate thread

# Design Patterns in Practice

**Model-View-Controller (MVC):**

**Model:**

- Ball, Paddle, Brick – data and logic
- Independent of GUI

**View:**

- Board.paintComponent() – rendering
- Model state presentation

**Controller:**

- MouseMotionListener – input handling
- BallEngine – state updates

**Observer Pattern:**

- Board observes mouse events
- Automatic notifications via listener

# Testing and Debugging

**Common problems:**

**Multithreading:**

- Race conditions – access synchronization
- Deadlocks – mutual blocking
- Use `synchronized` or `SwingUtilities`

**Collisions:**

- Ball "sinking" into objects
- Multiple detections of same collision
- Position correction after detection

**Performance:**

- FPS measurement
- Code profiling
- Collision loop optimization

# Best Practices

**Good programming practices:**

**Code:**

- Encapsulation – private fields, public methods
- Single Responsibility – one class, one purpose
- DRY (Don't Repeat Yourself) – avoid duplication

**GUI and multithreading:**

- All GUI operations in EDT
- Long operations in separate threads
- Use `SwingUtilities.invokeLater()`

**Documentation:**

- Javadoc for public methods
- Comments for complex logic
- README with launch instructions

# Project Extensions – Ideas

**Features to add:**

**Gameplay:**

- Lives system
- High scores (save to file)
- Two-player mode
- Special bricks (requiring multiple hits)

**Visual:**

- Brick destruction animations
- Particle effects
- Parallax background
- Menu and pause screens

**Technical:**

- Configuration from file (JSON/XML)
- Level editor
- Network multiplayer

# Summary – Techniques Used

**Concept overview:**

**Object-oriented programming:**

- Classes: Ball, Paddle, Brick, Board, BallEngine
- Inheritance: extends Ellipse2D.Float, Rectangle2D.Float, Thread, JPanel
- Polymorphism: Shape, MouseMotionListener
- Encapsulation: private fields, public methods

**Library classes:**

- Java2D: Shape, Graphics2D, Ellipse2D, Rectangle2D
- Swing: JFrame, JPanel, JComponent
- Collections: ArrayList
- Events: MouseMotionListener, MouseEvent

# Summary – Continued

**Multithreading:**

- Thread inheritance
- run() method – animation loop
- sleep() – frequency control
- EDT – Event Dispatch Thread
- SwingUtilities.invokeLater()

**2D Graphics:**

- paintComponent() – rendering
- Graphics2D – graphics context
- fill() – shape filling
- repaint() – schedule redraw

**Interactivity:**

- MouseMotionListener – mouse handling
- mouseMoved() – movement response
- MouseEvent – event information

# Key Lessons

**What we learned:**

1. **Technology integration:**
   - OOP + GUI + Graphics + Threading
   - Different Java packages cooperation
2. **Application architecture:**
   - Class division by responsibility
   - Object relationships
3. **Practical theory application:**
   - Inheritance for code reuse
   - Polymorphism for flexibility
   - Interfaces for contracts
4. **Problem solving:**
   - Collision detection
   - Thread synchronization
   - Performance optimization