

# Инструментарий: программные интерфейсы

Лекция 2



# Оглавление

7. Инструментарий: программные интерфейсы	3
В предыдущем разделе	3
7.1. Введение и результат	4
<b>7.2. Подготовка проекта</b>	<b>5</b>
7.2.1. Основное окно	5
7.2.2. Канва для рисования	5
7.2.3. Цикл отрисовки	7
7.2.4. Параметры отрисовки	9
<b>7.3. Рисуемые объекты</b>	<b>11</b>
7.3.1. Двумерный рисуемый объект (спрайт)	11
7.3.2. Конкретный рисуемый объект	12
<b>7.4. Интерфейсы</b>	<b>15</b>
7.4.1. Пример без интерфейсов	15
7.4.2. Понятие интерфейса	17
7.4.3. Реализация интерфейса	19
Рис. 4: Наследование интерфейсов	22
7.4.4. Вопросы для самопроверки	22
7.4.5. Применение интерфейса	23
7.4.6. Создание библиотечных классов	24
7.4.7. Особенности интерфейсов	27
<b>7.5. Анонимные классы</b>	<b>28</b>
<b>7.5.1. Понятие и применение</b>	<b>28</b>
7.5.2. Альтернативы	31
7.5.3. SOLID	33
7.5.4. Вопросы для самопроверки	33
<b>7.6. Исключения в графических интерфейсах пользователя</b>	<b>34</b>
<b>Практическое задание</b>	<b>37</b>
<b>Термины, определения и сокращения</b>	<b>37</b>

## 7. Инструментарий: программные интерфейсы

### В предыдущем разделе


В предыдущем разделе были рассмотрены графические интерфейсы пользователя.

- Создание окон,
- размещение компонентов,
- рисование элементов,
- обработка событий.

### В этом разделе

Будет рассмотрено понятие и принцип работы программных интерфейсов, ключевое слово `implements`. Реализация интерфейса, реализация по умолчанию, частичная реализация интерфейса, наследование и множественное наследование интерфейсов. Отдельно будет рассмотрен принцип создания так называемых адаптеров и анонимных классов. Знания в области применения графических фреймворков будут дополнены информацией о поведении исключений на интерфейсе пользователя.

- Интерфейс;
- `implements`;
- Анон. класс;
- Реализация;
- Адаптер;
- Рендеринг;
- SOLID;

 Обычно – теория и примеры. В этом разделе повествование будет построено от практики и плохого кода к хорошему коду и теоретическому обоснованию сделанного.

## 7.1. Введение и результат

В этом разделе важно не особенно обращать внимание на то, какие именно классы и методы используются, а внимательно следить за взаимодействием и отношениями объектов, потому что интерфейсы, о которых далее планируется говорить – это механизм упрощающий и универсализирующий взаимодействия объектов.

Поначалу, код может показаться непростым, но задача будет поставлена таким образом, что без программных интерфейсов не обойтись. Почему будет сложно? Несмотря на то что принципы ООП уже известны, нужно уметь их применять.



Рис. 1: Результат выполнения написанного кода

В результате работы с этим разделом будет создан некий небольшой демонстрационный набросок игрового двухмерного движка, без физики, с объектами и анимацией. На рисунке 1 изображено окно с кружками<sup>1</sup>. На окне ничего не обрабатывается и практически ничего не происходит. Для реализации задуманного понадобится: окно, которое будет взаимодействовать с операционной системой, канва, на которой будет происходить рисование, и объекты, которые будут нарисованы.

---

<sup>1</sup> [Видео с демонстрацией движения](#)

## 7.2. Подготовка проекта

### 7.2.1. Основное окно

Сложное окно не нужно: константы с размерами, координатами, и конструктор здесь же в основном методе. Самое важное сейчас это то, что окно – это объект с какими-то свойствами и каким-то поведением.

```
1 package ru.gb.jdk.two.online;
2
3 import javax.swing.*;
4
5 public class MainWindow extends JFrame {
6     private static final int POS_X = 400;
7     private static final int POS_Y = 200;
8     private static final int WINDOW_WIDTH = 800;
9     private static final int WINDOW_HEIGHT = 600;
10
11     private MainWindow() {
12         setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
13         setBounds(POS_X, POS_Y, WINDOW_WIDTH, WINDOW_HEIGHT);
14         setTitle("Circles");
15
16         setVisible(true);
17     }
18
19     public static void main(String[] args) {
20         new MainWindow();
21     }
22 }
```

Листинг 1: Основное окно

### 7.2.2. Канва для рисования

Для рисования будет использоваться компонент JPanel, наследнику панели будет дано название MainCanvas. Любой компонент фреймворка Swing может перерисовываться, вызывая метод `paintComponent()`. Для начала, в конструкторе панели для отработки связи компонентов следует делать что-то незначительное, например, менять цвет фона на синий.



Переопределив метод перерисовки панели не следует удалять вызов родительского метода, поскольку предполагается, что перерисовка панели происходит хорошо, но туда будет добавлена логика.

Также для универсализации дальнейших вызовов (удобства взаимодействия с канвой) следует добавить методы, возвращающие границы панели.

```

1 package ru.gb.jdk.two.online;
2
3 import javax.swing.*;
4 import java.awt.*;
5
6 public class MainCanvas extends JPanel {
7     MainCanvas() {
8         setBackground(Color.BLUE);
9     }
10
11     @Override
12     protected void paintComponent(Graphics g) {
13         super.paintComponent(g);
14     }
15
16     public int getLeft() { return 0; }
17     public int getRight() { return getWidth() - 1; }
18     public int getTop() { return 0; }
19     public int getBottom() { return getHeight() - 1; }
20 }

```

Листинг 2: Панель для канвы



Нет прямого запрета на написание логики будущего движка или игры в классе канвы, но это архитектурно неверное решение, ведь на канве, на которой происходит рисование, должно происходить только рисование. Подробнее в разделе 7.5.3

Исходя из принципа единой ответственности, принимается решение о том, что логика взаимодействия объектов будет описана в основном классе, а MainCanvas останется универсальным, чтобы иметь возможность в дальнейшем рисовать что угодно. Для этого следует описать в основном окне метод, который будет периодически вызываться канвой, например, `onDrawFrame()`. В нём будет описываться бизнес-логика. На начальном этапе, это два метода – `update()` который

будет изменять состояние приложения, и `render()`, который будет отдавать команды рисующимся компонентам.

```

1 private MainWindow() {
2     setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
3     setBounds(POS_X, POS_Y, WINDOW_WIDTH, WINDOW_HEIGHT);
4     setTitle("Circles");
5
6     MainCanvas canvas = new MainCanvas();
7     add(canvas);
8     setVisible(true);
9 }
10
11 public void onDrawFrame() {
12     update();
13     render();
14 }
15
16 private void update() { }
17 private void render() { }

```

Листинг 3: Отделение логики

### 7.2.3. Цикл отрисовки

Перерисовка канвы – это циклический процесс, и на каждой итерации `MainCanvas` должен вызывать метод `onDrawFrame()` основного класса. Для этого канве необходимо иметь ссылку на основное окно и внутри метода `paintComponent()` вызывается метод `controller.onDrawFrame()`.

```

1 public class MainCanvas extends JPanel {
2     private final MainWindow controller;
3
4     MainCanvas(MainWindow controller) {
5         setBackground(Color.BLUE);
6         this.controller = controller;
7     }
8
9     @Override
10    protected void paintComponent(Graphics g) {
11        super.paintComponent(g);
12        controller.onDrawFrame();
13    }

```

Листинг 4: Событие отрисовки

Далее, чтобы зациклить это действие, возможно два пути: самый простой – создать постоянно обновляющуюся канву, то есть в методе `paintComponent()` вызывать

repaint() но это полностью нагрузит одно из ядер процессора только отрисовкой окна.

```

1 public class MainCanvas extends JPanel {
2     private final MainWindow controller;
3
4     MainCanvas(MainWindow controller) {
5         setBackground(Color.BLUE);
6         this.controller = controller;
7     }
8
9     @Override
10    protected void paintComponent(Graphics g) {
11        super.paintComponent(g);
12        controller.onDrawFrame();
13        repaint();
14    }

```

Листинг 5: Цикл отрисовки

Второй путь – любой поток возможно заставить какое-то время поспать, для этого вызывается статический метод класса Thread, принимающий в качестве аргумента количество миллисекунд, которое поток должен обязательно поспать. Это даст FPS<sup>2</sup> близкий к 60, приемлемый для применения в цифровой технике.

```

1 @Override
2 protected void paintComponent(Graphics g) {
3     super.paintComponent(g);
4     controller.onDrawFrame();
5     try {
6         Thread.sleep(16);
7     } catch (InterruptedException e) {
8         throw new RuntimeException(e);
9     }
10    repaint();
11 }

```

Листинг 6: Снятие нагрузки с процессора

В результате создан бесконечный цикл отрисовки, аналогичный циклу do-while, который сам себя заставляет крутиться с некоторой периодичностью и на каждой итерации сообщает контроллеру, что прошло около одной шестидесятой секунды.

<sup>2</sup> FPS, Frames per second – (англ. кадров в секунду) количество сменяемых кадров за единицу времени в кинематографе, телевидении, компьютерной графике и т. д.



```

1 private MainWindow() {
2     setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
3     setBounds(POS_X, POS_Y, WINDOW_WIDTH, WINDOW_HEIGHT);
4     setTitle("Circles");
5
6     MainCanvas canvas = new MainCanvas(this);
7     add(canvas);
8     setVisible(true);
9 }

```

Листинг 7: Изменения в основном окне

В код, вызывающий конструктор канвы, также требуется внести незначительные изменения. В канву необходимо передать ссылку на основное окно, на котором она находится, для этого используется ключевое слово `this`.



Такой способ применения ключевого слова `this` ещё пригодится в этом разделе.

## 7.2.4. Параметры отрисовки

Метод `onDrawFrame()` будет обновлять сцену и заставляя объекты на ней рисовать самих себя (решать сцену). Для обновления сцены, привязанного ко времени физического мира необходимо знать дельту времени, то есть период времени, прошедший с появления предыдущего кадра.



Писать логику обновления, исходя из частоты кадра, или из того, что канва «спит» 16 миллисекунд – очень сомнительная опора, потому что поток **гарантированно** ждёт 16 миллисекунд. При этом, сколько будут выполняться остальные действия – неизвестно, так как отрисовка происходит не через фиксированные промежутки времени, а по очереди сообщений окна и под влиянием множества других факторов.

Метод `onDrawFrame()` должен принимать от канвы ряд параметров и распределять их по методам обновления и рендеринга.

```

1 public void onDrawFrame(MainCanvas canvas, Graphics g, float deltaTime) {
2     update(canvas, deltaTime);
3     render(canvas, g);
4 }
5
6 private void update(MainCanvas canvas, float deltaTime) {
7
8 }
9 private void render(MainCanvas canvas, Graphics g) {
10
11 }

```

Листинг 8: Параметры метода обновления

При вычислении дельты времени важно привести все единицы измерения к единому и привычному времени, например, к секундам. Скоростью в этом случае будет «пиксель в секунду» и из метода будет отдаваться время в секундах. При обращении к контроллеру передаётся также ссылка на текущий объект канвы и объект графики. Пока самое главное, что нужно понять об этих двух объектах – канва считает время в физическом мире и постоянно перерисовывает себя, сообщая об этом факте основному окну, а основное окно на этот факт как-то реагирует<sup>3</sup>.

```

1 private long lastFrameTime;
2
3 MainCanvas(MainWindow controller) {
4     this.controller = controller;
5     lastFrameTime = System.nanoTime();
6 }
7
8 @Override
9 protected void paintComponent(Graphics g) {
10     super.paintComponent(g);
11     try {
12         Thread.sleep(16);
13     } catch (InterruptedException e) {
14         throw new RuntimeException(e);
15     }
16     float deltaTime = (System.nanoTime() - lastFrameTime) * 0.00000001f;
17     controller.onDrawFrame(this, g, deltaTime);
18     lastFrameTime = System.nanoTime();
19     repaint();
20 }

```

Листинг 9: Вычисление дельты времени между кадрами канвы

<sup>3</sup> ООП вокруг этих событий тоже важно хорошо понимать – объекты передают ссылки друг на друга и вызывают друг у друга методы.

Приложение будет рисовать какие-то объекты, и не важно, будут ли это кружки, квадратики, картинки, человечки или какие-то другие объекты. Важно, чтобы у программы было описано поведение этих объектов<sup>4</sup>.

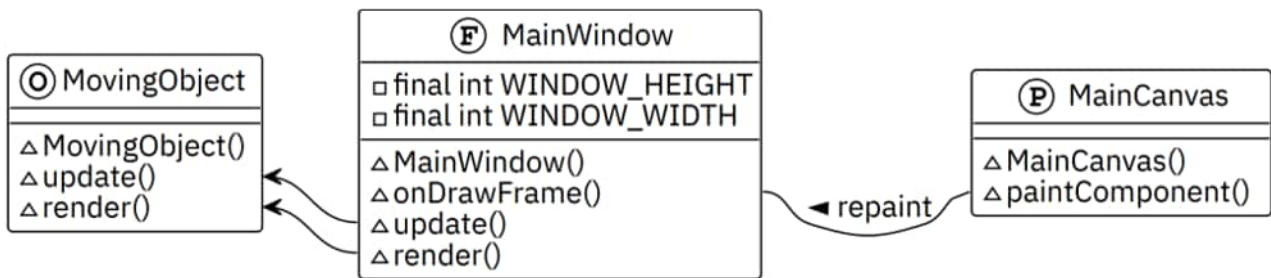


Рис. 2

## 7.3. Рисуемые объекты

### 7.3.1. Двумерный рисуемый объект (спрайт)

Класс `Sprite` описывает общие для всех рисуемых объектов в программе поведение и свойства. В графических фреймворках часто начало координат находится в верхнем левом или нижнем левом углу. Однако, очень часто, когда пишутся какие-то игры или другие приложения с использованием графики в качестве координат используется центр объекта. То есть необходимо условиться, что `X` и `Y` – это центр любого визуального объекта на канве. И, следовательно, удобно хранить не длину с шириной, а половину длины и половину ширины. А границы объекта, соответственно, будут отдаваться через геттеры и сеттеры. Дополнительно следует указать спрайту, что он умеет обновляться и рендериться, а его наследники уже смогут самостоятельно решать, как именно они хотят это делать.

<sup>4</sup> [Исходный код PlantUML](#)

```

1 package ru.gb.jdk.two.online;
2
3 import java.awt.*;
4
5 public abstract class Sprite {
6     protected float x;
7     protected float y;
8     protected float halfWidth;
9     protected float halfHeight;
10
11     protected float getLeft() { return x - halfWidth; }
12     protected void setLeft(float left) { x = left + halfWidth; }
13     protected float getRight() { return x + halfWidth; }
14     protected void setRight(float right) { x = right - halfWidth; }
15     protected float getTop() { return y - halfHeight; }
16     protected void setTop(float top) { y = top + halfHeight; }
17     protected float getBottom() { return y + halfHeight; }
18     protected void setBottom(float bottom) { y = bottom - halfHeight; }
19
20     protected float getWidth() { return 2f * halfWidth; }
21     protected float getHeight() { return 2f * halfHeight; }
22
23     void update(MainCanvas canvas, float deltaTime) { }
24     void render(MainCanvas canvas, Graphics g) { }
25 }

```

Листинг 10: Абстрактный рисуемый объект – спрайт

## 7.3.2. Конкретный рисуемый объект

Инстанцировать абстрактный класс нельзя, поэтому, нужно создать класс шарика, который будет перемещаться по экрану. В конструкторе шарика задаются случайные размеры с определённым разбросом. Чтобы не усложнять пример отдельными объектами, описывающими физику мира, непосредственно объекту шарика будут заданы скорости по осям X и Y, и цвет.

```

1 package ru.gb.jdk.two.online;
2
3 import java.awt.*;
4 import java.util.Random;
5
6 public class Ball extends Sprite {
7     private static Random rnd = new Random();
8     private final Color color;
9     private float vX;
10    private float vY;
11
12    Ball() {
13        halfHeight = 20 + (float) (Math.random() * 50f);
14        halfWidth = halfHeight;
15        color = new Color(rnd.nextInt());
16        vX = 100f + (float) (Math.random() * 200f);
17        vY = 100f + (float) (Math.random() * 200f);
18    }
19
20    @Override
21    void update(MainCanvas canvas, float deltaTime) {
22        x += vX * deltaTime;
23        y += vY * deltaTime;
24
25        if (getLeft() < canvas.getLeft()) {
26            setLeft(canvas.getLeft());
27            vX = -vX;
28        }
29        if (getRight() > canvas.getRight()) {
30            setRight(canvas.getRight());
31            vX = -vX;
32        }
33        if (getTop() < canvas.getTop()) {
34            setTop(canvas.getTop());
35            vY = -vY;
36        }
37        if (getBottom() > canvas.getBottom()) {
38            setBottom(canvas.getBottom());
39            vY = -vY;
40        }
41    }
42
43    @Override
44    void render(MainCanvas canvas, Graphics g) {
45        g.setColor(color);
46        g.fillOval((int) getLeft(), (int) getTop(),
47            (int) getWidth(), (int) getHeight());
48    }
49 }

```

Листинг 11: Рисуемый объект

В классе шарика переопределяются методы обновления и рендеринга. Самый простой рендер – объекту графики задаётся цвет текущего шарика и вызывается метод `fillOval()`, которому передаются левая и верхняя координаты, ширина и

высота. Несмотря на то, что объекты содержат поля типа float, работа происходит с пиксельной системой координат, а значит необходимо переводить в целые числа<sup>5</sup>

В методе обновления к текущим координатам шарика прибавляется расстояние, которое должен был преодолеть шарик за то время пока канва спала и рендерилась.

$$ball(x_{new}, y_{new}) = ball(x + vx * \delta t, y + vy * \delta t).$$

Дополнительно, обрабатываются отскоки от границ панели, то есть описаны четыре условия, что при достижении границы меняется направление вектора. В основном классе делается очень прямолинейно – создаётся массив из спрайтов, способный удерживать десять шариков. В методе обновления каждый шарик из массива необходимо попросить обновиться, а в методе рендеринга – дать команду на отрисовку.



Реализация обновления и отрисовки остаётся самим объектам, то есть инкапсулируется в них. Только каждый объект сам по себе знает, как именно ему обновляться с течением времени, и как рисоваться, а основной экран управляет на более высоком уровне – на какой канве, когда и что рисовать.

В конструкторе добавляется простой цикл инициализирующий приложение десятью шариками.

<sup>5</sup> Такой способ, конечно же, не подходит для реальных проектов, там необходимо всё сразу переводить в «мировые координаты» (например принять центр экрана за 0, верхний-левый угол за -1 нижний-правый за 1, как это делается в OpenGL) чтобы рендерить экраны.

```

1 private final Sprite[] sprites = new Sprite[10];
2
3 private MainWindow() {
4     // ...
5     for (int i = 0; i < sprites.length; i++) {
6         sprites[i] = new Ball();
7     }
8     // ...
9 }
10
11 private void update(MainCanvas canvas, double deltaTime) {
12     for (int i = 0; i < sprites.length; i++) {
13         sprites[i].update(canvas, deltaTime);
14     }
15 }
16
17 private void render(MainCanvas canvas, Graphics g) {
18     for (int i = 0; i < sprites.length; i++) {
19         sprites[i].render(canvas, g);
20     }
21 }

```

Листинг 12: Управление объектами приложения

Напомню, что самое главное, что необходимо понять из этого приложения – это взаимодействия и взаимовлияния объектов. Наследование, полиморфизм, инкапсуляция поведений и свойств.

## 7.4. Интерфейсы

### 7.4.1. Пример без интерфейсов



Начать разговор об интерфейсах я решил с создания отдельного класса фона, но сразу столкнулся с необходимостью думать головой...

На первый взгляд, логично было бы предположить, что фон – это спрайт, имеющий прямоугольную форму и всегда рисующийся первым. Но, есть затруднения, связанные с таким подходом: при изменении размеров окна фон тоже желательно изменить в размерах, а это лишние слушатели и десятки строк кода, поэтому при

отрисовке объекта фона гораздо проще будет дать команду канве на изменение фона.

```

1 package ru.gb.jdk.two.online;
2
3 import java.awt.*;
4
5 public class Background extends Sprite {
6     private float time;
7     private static final float AMPLITUDE = 255f / 2f;
8     private Color color;
9
10    @Override
11    public void update(MainCanvas canvas, float deltaTime) {
12        time += deltaTime;
13        int red = Math.round(AMPLITUDE + AMPLITUDE * (float) Math.sin(time * 2f));
14        int green = Math.round(AMPLITUDE + AMPLITUDE * (float) Math.sin(time * 3f));
15        int blue = Math.round(AMPLITUDE + AMPLITUDE * (float) Math.sin(time));
16        color = new Color(red, green, blue);
17    }
18
19    @Override
20    public void render(MainCanvas canvas, Graphics g) {
21        canvas.setBackground(color);
22    }
23 }
24
25 // MainCanvas
26 sprites[0] = new Background();
27 for (int i = 1; i < sprites.length; i++) {
28     sprites[i] = new Ball();
29 }

```

Листинг 13: Фон, как наследник спрайта и изменения в основном классе

Цвет фона меняется синусоидально по каждому из трёх компонентов цвета, поэтому изменение происходит плавно (рис. 3). Для реализации фона от спрайта, фактически, нужно только поведение, а свойства не нужны. Но и отказываться от наследования не очень правильно, потому что тогда не получится фон единообразно в составе массива спрайтов обновлять. Эти факты напрямую намекают на унификацию поведения – на интерфейс.



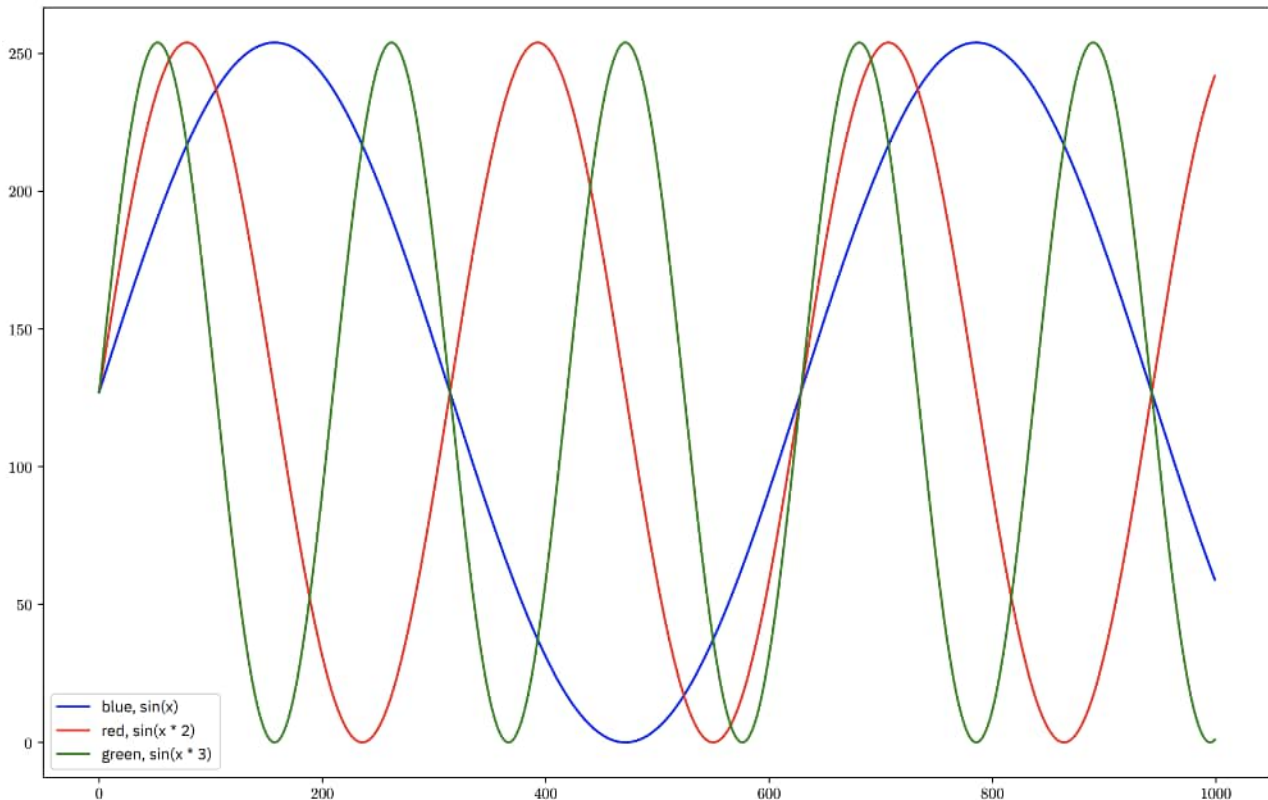


Рис. 3: График изменения значений компонентов цвета

## 7.4.2. Понятие интерфейса



Механизм наследования очень удобен, но он имеет свои ограничения. В частности, в языке Java допустимо наследование только от одного класса, в отличие, например, от языка C++, где имеется множественное наследование.

В языке Java проблему отсутствия множественного наследования частично позволяют решить интерфейсы. Интерфейсы определяют некоторый функционал, не имеющий конкретной реализации, который затем реализуют классы, применяющие эти интерфейсы. Один класс может применить к себе множество интерфейсов. Правильно говорить «реализовать интерфейс». Интерфейс можно очень примерно представить как очень абстрактный класс. Интерфейс – это описание методов.

💡 Интерфейс – это описание способов взаимодействия с объектом. Интерфейсы определяют функционал, не имеющий конкретной реализации.

Примером интерфейса в реальной жизни может быть интерфейс управления автомобилем, интерфейс взаимодействия с компьютером или интерфейс USB, так, компьютеру не важно, что именно находится по ту сторону провода – флеш накопитель, веб-камера или мобильный телефон, а важно, что компьютер умеет работать с интерфейсом USB, отправлять туда байты или получать. Потоки вводавывода, которые были изучены – это тоже своего рода интерфейс, соединяющий не важно какого отправителя, например, программный код и не важно какого получателя, например, файл. Интерфейсы объявляются также, как классы, и могут иметь очень похожую на класс структуру, то есть быть вложенным или внутренним. Чаще всего каждый отдельный интерфейс описывают в отдельном файле, также как класс, но используя ключевое слово `interface`. Ниже показаны примеры интерфейсов, человек и бык, в которых описаны методы «ходить» и «издавать звуки».

Все методы во всех интерфейсах всегда публичные, и в классическом варианте (до Java 1.8) не имеют никакой реализации. Поскольку все методы всегда публичные, то этот модификатор принято не писать.

```

1 package ru.gb.jdk.two.online.samples;
2
3 public interface Human {
4     public void walk();
5     public void talk();
6 }
7
8 package ru.gb.jdk.two.online.samples;
9
10 public interface Bull {
11     void walk();
12     void talk();
13 }
```

Листинг 14: Примеры интерфейсов

### 7.4.3. Реализация интерфейса

Продолжая учебный пример: созданы классы мужчина и бык. Класс мужчины реализовывает интерфейс человека, а класс быка – быка.



Для реализации интерфейса необходимо переопределить все его методы, либо сделать класс абстрактным.

Множественного наследования нет, но существует возможность реализовать любое количество интерфейсов.

```

1 package ru.gb.jdk.two.online.samples;
2
3 public class Man implements Human {
4     @Override
5     public void walk() {
6         System.out.println("Walks on two feet");
7     }
8
9     @Override
10    public void talk() {
11        System.out.println("Talks meaningful words");
12    }
13 }
14
15 package ru.gb.jdk.two.online.samples;
16
17 public class Ox implements Bull {
18     @Override
19     public void walk() {
20         System.out.println("Walks on hooves");
21     }
22
23     @Override
24     public void talk() {
25         System.out.println("MooOooOoooO0oo");
26     }
27 }

```

Листинг 15: Классы, реализующие интерфейс

Одним из самых удобных следствий применения интерфейсов является возможность объявлять не только классы и создавать объекты, но и создать идентификторы, которые ссылаются на объект, реализующий интерфейс. То есть, по идентификатору типа интерфейса могут лежать абсолютно не связанные между собой объекты, главное, чтобы они реализовывали интерфейс. При этом

сохраняется возможность работать с методами интерфейса которые могут быть для разных классов по-разному реализованы. Это иная форма изученного ранее **полиморфизма**.

```

1 package ru.gb.jdk.two.online.samples;
2
3 public class Main {
4     public static void main(String[] args) {
5         Man man0 = new Man(); //class Man
6         Ox ox0 = new Ox(); // class Ox
7         Human man1 = new Man(); // interface Human
8         Bull ox2 = new Ox(); // interface Bull
9     }
10 }

```

Листинг 16: Интерфейсные переменные

Для демонстрации ещё одного способа применения интерфейсов, будет описан класс минотавра<sup>6</sup>, реализовывающий интерфейсы человека и быка своим собственным способом, а именно, ходил на ногах человека, но не мычал, как бык, а загадывал загадки.



При использовании интерфейсов важно то, что классы не связаны между собой наследованием, а обращение к ним единообразно.

Интересно то, что в программе таким образом появляется возможность обратиться к минотавру не только как к человеку, но и как к быку, то есть гипотетически, можно создать некоторого Тесея, управляющего большим количеством минотавров.

<sup>6</sup> мифический персонаж с телом человека и головой быка

```

1 package ru.gb.jdk.two.online.samples;
2
3 public class Main {
4     private static class Minotaurus implements Human, Bull {
5         @Override public void walk() {
6             System.out.println("Walks on two legs");
7         }
8
9         @Override public void talk() {
10             System.out.println("Asks you a riddle");
11         }
12     }
13     public static void main(String[] args) {
14         Bull minos0 = new Minotaurus();
15         Human minos1 = new Minotaurus();
16         Minotaurus minos = new Minotaurus();
17         Human man1 = new Man();
18         Bull ox2 = new Ox();
19         Bull[] allBulls = {ox2, minos0, minos};
20         Human[] allHumans = {man1, minos, minos1};
21     }
22 }

```

Листинг 17: Реализация множества интерфейсов

Также важно, что в интерфейсах разрешено наследование. То есть, один интерфейс может наследоваться от другого интерфейса, соответственно, при реализации такого, наследующего интерфейса, необходимо переопределять не только методы интерфейса, но и методы всех его родителей, также, как если бы происходило переопределение методов абстрактного класса.



Следует обратить особое внимание на то, что в интерфейсах разрешено множественное наследование.

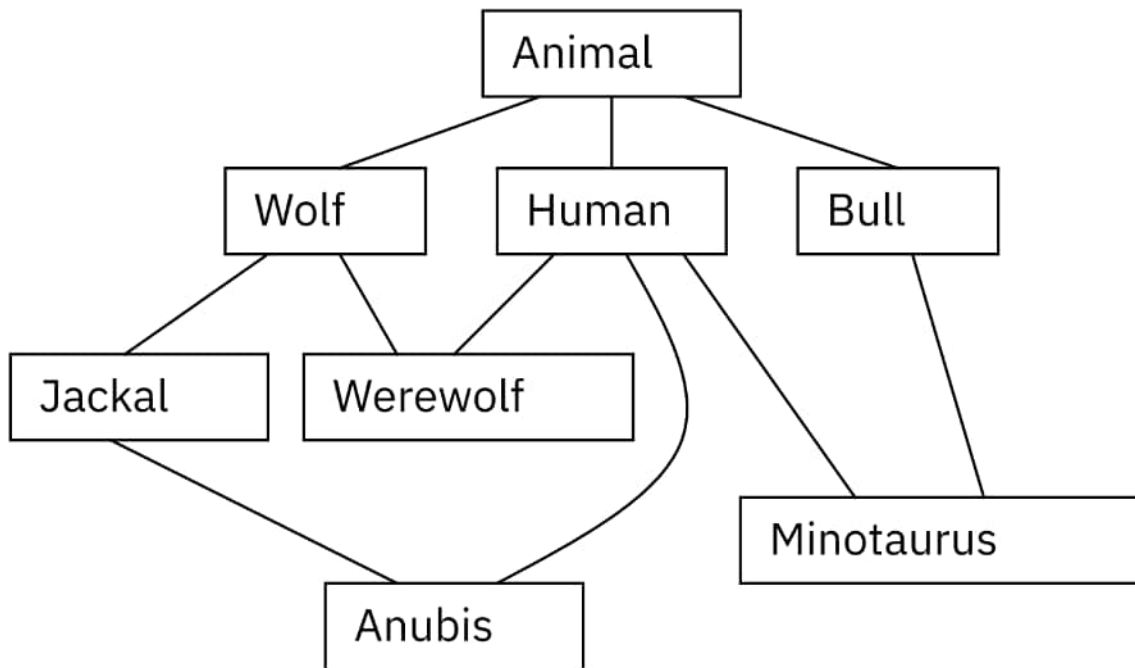


Рис. 4: Наследование интерфейсов

## 7.4.4. Вопросы для самопроверки

1. Программный интерфейс – это:

- (a) окно приложения в ОС;
- (b) реализация методов объекта;
- (c) объявление методов, реализуемых в классах.

2. Интерфейсы нужны для:

- (a) компенсации отсутствия множественного наследования;
- (b) отделения API и реализации;
- (c) оба варианта верны.

3. Интерфейсы позволяют:

- (a) удобно создавать новые объекты, не связанные наследованием;
- (b) единообразно обращаться к методам объектов, не связанных наследованием;
- (c) полностью заменить наследование.

## 7.4.5. Применение интерфейса

В описанном ранее примере интерфейс помогает решить проблему единообразия поведения спрайтов и фона, при их различии в свойствах. То есть, сложилась ситуация в которой существует необходимость хранить в одном массиве объекты со схожим поведением, но наследовать их друг от друга не совсем логично.

```

1 package ru.gb.jdk.two.online;
2
3 import java.awt.*;
4
5 public interface Interactable {
6     void update(MainCanvas canvas, float deltaTime);
7     void render(MainCanvas canvas, Graphics g);
8 }

```

Листинг 18: Интерфейс обновляемого и рисуемого объекта

В качестве решения описан интерфейс `Interactable`, содержащий методы обновления и рендеринга без реализации.



Если описывать ещё более гибкое приложение, нужно создавать два интерфейса, `Updatable` и `Renderable`, чтобы иметь возможность отделить рисуемые объекты от обновляемых.

В данном случае интерфейс описывает объекты, которые должны уметь рисоваться и обновляться. В спрайте и фоне интерфейс реализуется. При этом получается, то фон никак не связан со спрайтом, но при этом, оба умеют рисоваться и обновляться, благодаря интерфейсу. Далее, при смене массива спрайтов на массив интерактивностей приложение не сломается.

```

1 public abstract class Sprite implements Interactable
2
3 public abstract class Background implements Interactable

```

Листинг 19: Реализация интерфейса объектами приложения

## 7.4.6. Создание библиотечных классов

Если обратить внимание на развитие повествования по курсу, можно заметить, что сначала произошёл выход за пределы одного метода (методы, помимо `main`), потом за пределы одного класса (классы котиков, собачек, и т.д.), затем за пределы одного пакета (логическое разделение классов), за пределы программного кода (поток ввода-вывода), а теперь код, который возможно использовать несколькими программами.

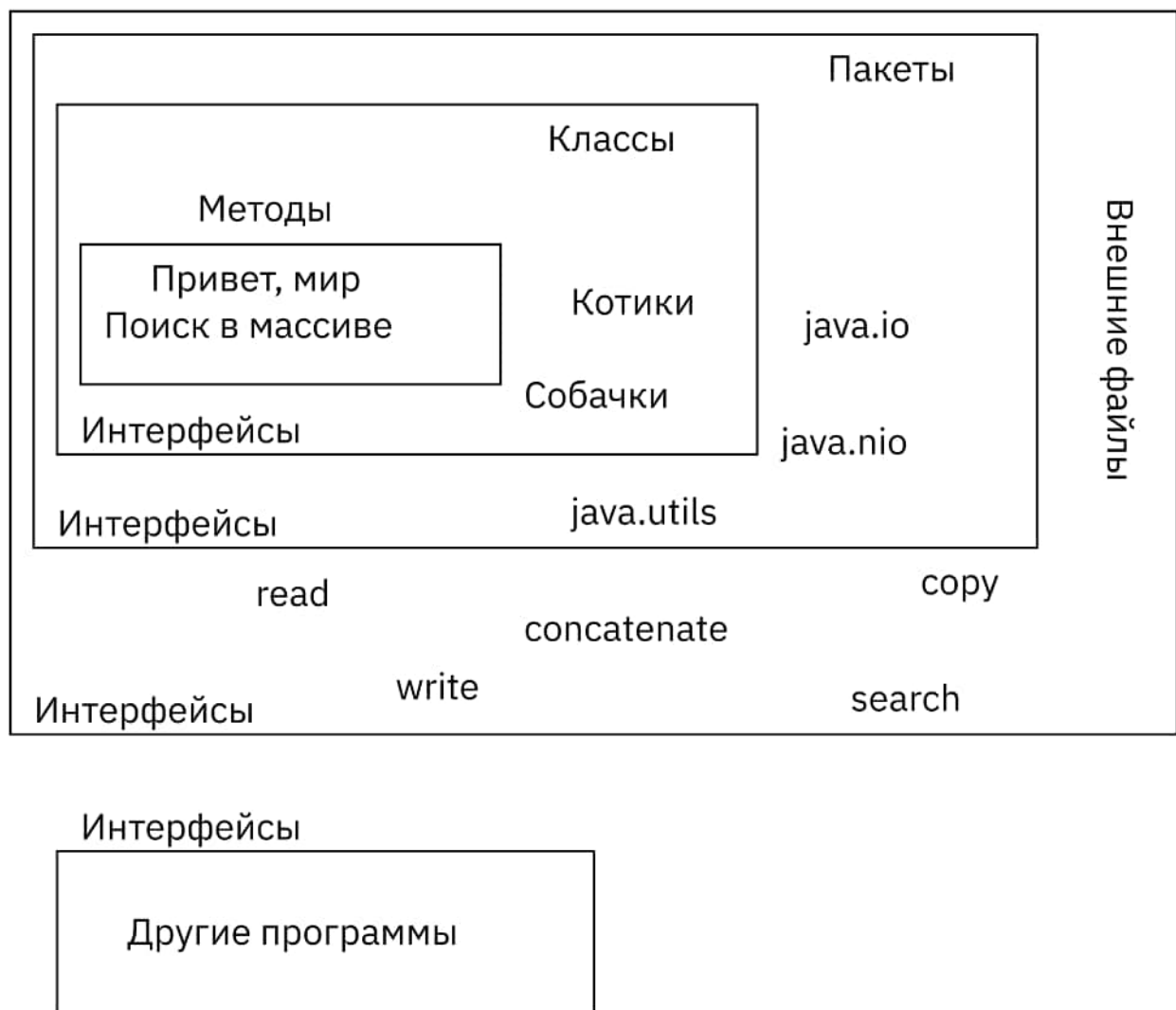
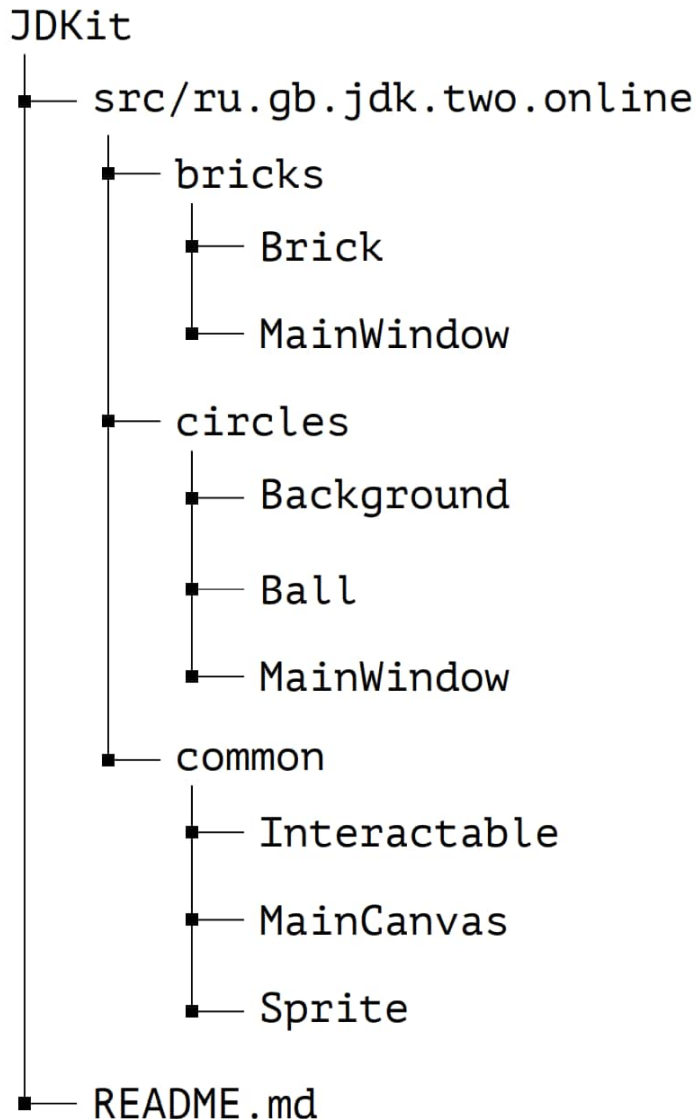


Рис. 5: Применение интерфейсов для отделения единиц компиляции

У классов канвы и спрайта, а также у интерфейса, нет никакой специфики, они ничего не «знают» о том, какие объекты существуют в программе и как эти объекты взаимодействуют между собой. Эти классы и интерфейс применимы, по сути, где угодно, не только в этой конкретной программе с этими конкретными классами. С




использованием таких общих фрагментов кода становится возможным достаточно быстро написать вторую игру: новый пакет, новый класс, скопированный код от основного окна шариков. А полностью копировать спрайты и интерфейсы не целесообразно. Сделав правильное дробление по пакетам становится очевидно, что существует общий библиотечный пакет, и какие-то приложения с конкретными реализациями.



В общий пакет классы скопировались без проблем, шарики перенесли с минимальными изменениями – только публичные модификаторы понадобились. В главном окне с будущими летающими квадратами создан аналогичный конструктор, размеры, положение. Но общей канвой воспользоваться невозможно. Это происходит, потому что канва может принимать в качестве параметра в

конструкторе только основное окно из пакета кружочков. И далее канва уже у класса с кружочками вызывает метод `onDrawFrame()`.

 Привязка к классу ограничивает возможности

Решение на поверхности – использование интерфейсов. Необходимо написать интерфейс, который может по смыслу называться, например, `CanvasRepaintListener`, который будет уметь ожидать от канвы вызов метода и реализовывать его.

```

1 package ru.gb.jdk.two.online.common;
2
3 import java.awt.*;
4
5 public interface CanvasRepaintListener {
6     void onDrawFrame(MainCanvas canvas, Graphics g, float deltaTime);
7 }

```

Листинг 20: Интерфейс слушателя событий канвы

Такой интерфейс логично создавать в общем пакете и переписать канву так, чтобы она принимала на вход не класс, а объект, реализующий интерфейс. Далее, оба слушателя реализуются через интерфейс.

```

1 public class MainCanvas extends JPanel {
2     private final CanvasRepaintListener controller;
3     private long lastFrameTime;
4
5     public MainCanvas(CanvasRepaintListener controller) {
6
7         this.controller = controller;
8         lastFrameTime = System.nanoTime();
9     }

```

Листинг 21: Использование интерфейса на канве

Интерфейс может быть реализован классом, а окна приложений – это тоже классы. Это позволяет не только наследоваться от классов фреймворка Swing, но и реализовывать интерфейсы, описанные внутри приложения. Следовательно, окно без изменений продолжает передавать себя в конструктор, а метод интерфейса уже реализован. Чтобы подчеркнуть, что это реализация интерфейса – дописана аннотация `@Override`.

```

1 public class MainWindow extends JFrame implements CanvasRepaintListener {
2     private MainWindow() { ... }
3
4     @Override
5     public void onDrawFrame(MainCanvas canvas, Graphics g, float deltaTime) { ... }

```

Листинг 22: Интерфейс обновляемого и рисуемого объекта

## 7.4.7. Особенности интерфейсов

Интерфейсы были значительно переработаны в Java 1.8, было добавлено довольно много механизмов, об одном из которых нельзя не сказать. Реализация интерфейсов по умолчанию. Пример будет построен на основе тех интерфейсов, которые уже написаны – человек и бык. Очевидно, что именно у этих интерфейсов возможны реализации по умолчанию, например, для действия «ходить»: человек ходит на двух ногах, а бык на четырёх копытах. Для описания реализации по умолчанию используется ключевое слово `default`. Если написать реализацию, но не использовать данное ключевое слово, произойдёт ошибка компиляции (или среда разработки укажет на ошибочность такой конструкции).

```

1 package ru.gb.jdk.two.online.samples;
2
3 public interface Human {
4     default void walk() {
5         System.out.println("Walks on two feet");
6     }
7
8     public void talk();
9 }
10
11 package ru.gb.jdk.two.online.samples;
12
13 public interface Bull {
14     void walk() { // compile time error
15         System.out.println("Walks on four hooves");
16     }
17     void talk();
18 }

```

Листинг 23: Реализация по умолчанию

Первое, и самое очевидное следствие использования реализации по умолчанию – отсутствие необходимости переопределять все методы в классах, реализующих эти интерфейсы, что делает интерфейс, в свою очередь, чуть более похожим на класс. Реализованные по умолчанию интерфейсы могут задействовать созданные в этом интерфейсе поля, а наличие в интерфейсе полей делает его ещё более похожим на

класс. Все поля в интерфейсах статические и неизменяемые, а если заменить публичный модификатор доступа на другой – будет ошибка компиляции.

```
1 public interface Bull {  
2     public static final int amount = 2;  
3     default void walk() {  
4         System.out.println("Walks on " + amount + " hooves");  
5     }  
6     void talk();  
7 }
```

Листинг 24: Использование полей в интерфейсах

## 7.5. Анонимные классы

### 7.5.1. Понятие и применение

Программные интерфейсы открывают перед разработчиком широчайшие возможности по написанию более выразительного кода. Одна из наиболее часто используемых возможностей – анонимные классы. Класс – это новый тип данных для программы. Классы бывают вложенными и внутренними. Внутренние классы – это классы, которые пишутся внутри других классов, которые в свою очередь описаны в файле. А также вложенные или локальные классы, которые возможно объявлять непосредственно в методах, и работать с ними, как с обычными классами. Анонимный класс, что довольно очевидно - это класс без названия. Далее приводится пример создания интерфейса `MouseListener` и описания в нём методов `mouseUp()`, `mouseDown()`. В основной части программы описан класс, реализующий этот интерфейс, то есть переопределяющий все его методы. Далее в методе `main()` создаётся экземпляр этого класса и появляется возможность использовать его методы.

```
1 public interface MouseListener {  
2     void mouseUp();  
3     void mouseDown();  
4 }  
5  
6 private static class MouseAdapter implements MouseListener {  
7     @Override public void mouseUp() { }  
8     @Override public void mouseDown() { }  
9 }  
10  
11 public static void main(String[] args) {  
12     MouseAdapter m = new MouseAdapter();  
13     m.mouseDown();  
14     m.mouseUp();  
15 }
```

Листинг 25: Способ использования API через именованный класс

Очень часто, элементы управления (кнопки, события входящих датчиков (клавиатура, мышка), сеть требуют на вход каких-то обработчиков собственных данных, которые будут слушать конкретный источник данных, отлавливать события и знать что делать. Это делается через интерфейсы. С точки зрения программы, создаётся некий метод, например, метод добавления к кнопке слушателя. Далее, если в элемент управления в качестве слушателя передаётся какой-то объект, реализующий нужный интерфейс, то этот объект начнёт ловить события и как-то их обрабатывать.

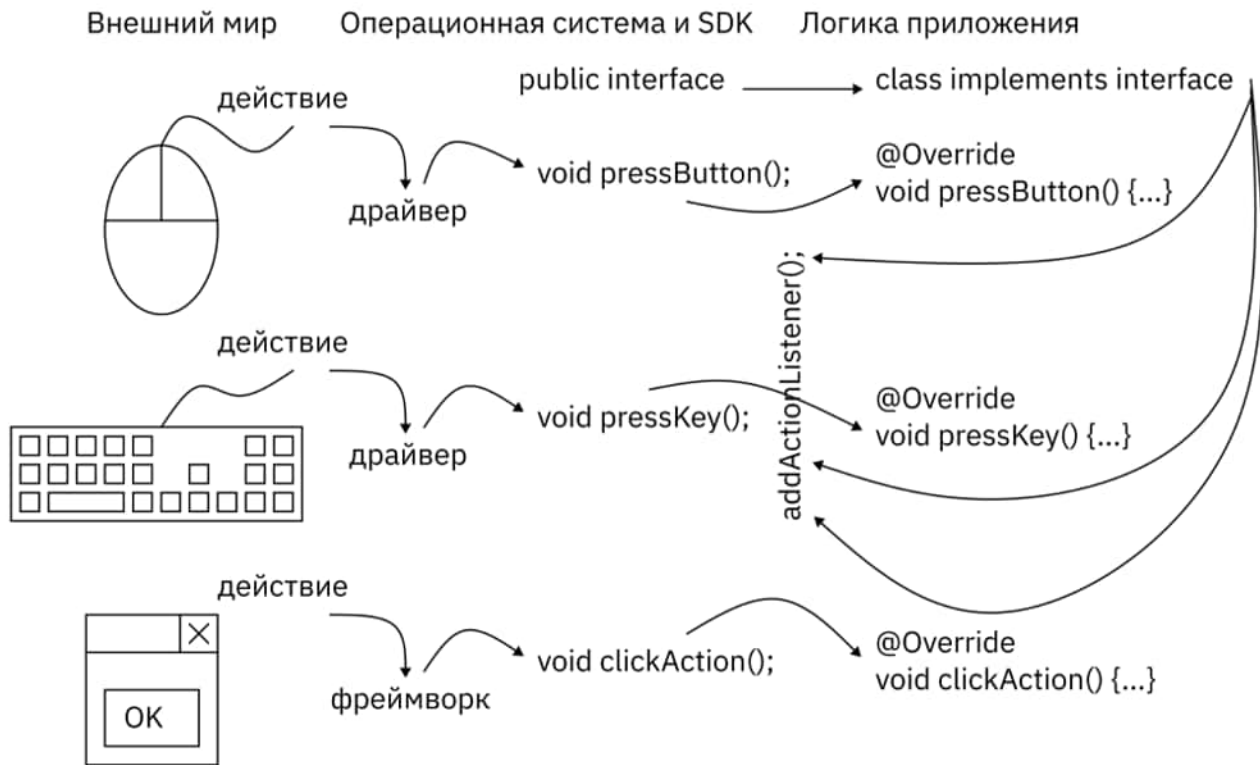


Рис. 6: Применение слушателей событий

Для полноценного примера (листинг 26) следует описать метод, принимающий на вход `MouseListener` (строка 1). И передать в метод объект (строка 13), предполагая, что внутри объекта для данного конкретного случая описано, как именно должна вести себя программа, когда кто-то нажал или отпустил кнопку мышки.

Допустим, есть параметр метода `MouseListener m` и необходимость создать туда некоторый экземпляр, который реализует этот интерфейс. Но есть класс, который это делает: `MouseAdapter`. Проще будет создать экземпляр адаптера, но без идентификатора (строка 14).

Часто такие классы создаются не просто без названия экземпляра, но и вовсе без имени, прямо в аргументе методов. Действительно, зачем классу имя, если он будет использован только один раз и только в этом методе для создания одного единственного объекта? Класс `MouseAdapter` идеально выполняет критерий **S** из принципов **SOLID** – **Single Responsibility**, делает только одно полезное дело – реализует интерфейс `MouseListener`. Но раз дело только одно – можно его выполнять и без размышлений о названии класса. Для создания интерфейсной переменной есть немного необычный синтаксис, на 15 строке. Получается что создаётся один экземпляр анонимного класса, который реализует интерфейс `MouseListener`. И созданный здесь же экземпляр данного класса кладётся в идентификатор.

Возможно также не создавать интерфейсный идентификатор, а сразу передать реализующий экземпляр в аргумент метода. Получится, что в метод передаётся новый экземпляр анонимного класса который реализует интерфейс слушателя, и тут же даётся описание этого класса, в котором переопределяются соответствующие методы (строка 20).

Ещё раз: **анонимные классы** – это классы, не имеющие названия и реализующие какой-то интерфейс.

```

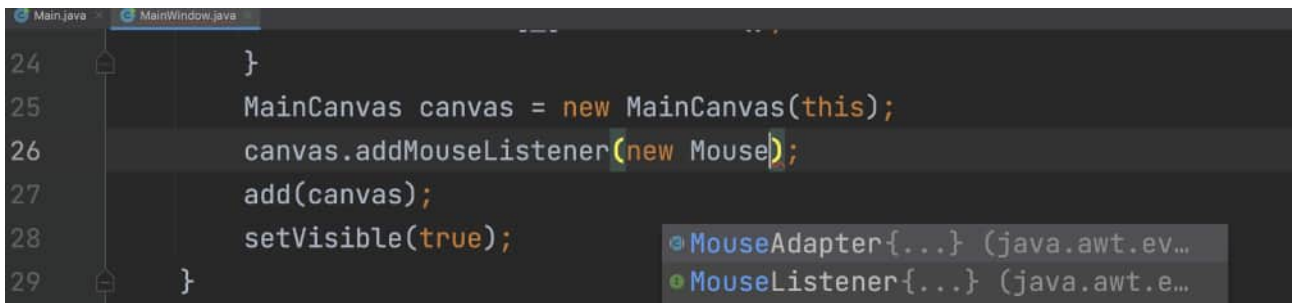
1 private static void addMouseListener(MouseListener l) {
2     l.mouseDown();
3
4     l.mouseUp();
5 }
6 private static class MouseAdapter implements MouseListener {
7     @Override public void mouseUp() { }
8     @Override public void mouseDown() { }
9 }
10
11 public static void main(String[] args) {
12     MouseAdapter m = new MouseAdapter();
13     addMouseListener(m);
14     addMouseListener(new MouseAdapter());
15     MouseListener l = new MouseListener() {
16         @Override public void mouseUp() { }
17         @Override public void mouseDown() { }
18     };
19     addMouseListener(l);
20     addMouseListener(new MouseListener() {
21         @Override public void mouseUp() { }
22         @Override public void mouseDown() { }
23 });

```

Листинг 26: Возможные способы создания обработчиков событий

## 7.5.2. Альтернативы

Существует возможность избежать использования анонимных классов и реализации сложных интерфейсов. Например, использовать адаптеры. Для панели на которой рисовались летающие шарики (7.2.2) существует слушатель и метод добавления реализации слушателя мышки, очень похожий на созданный в учебных целях.



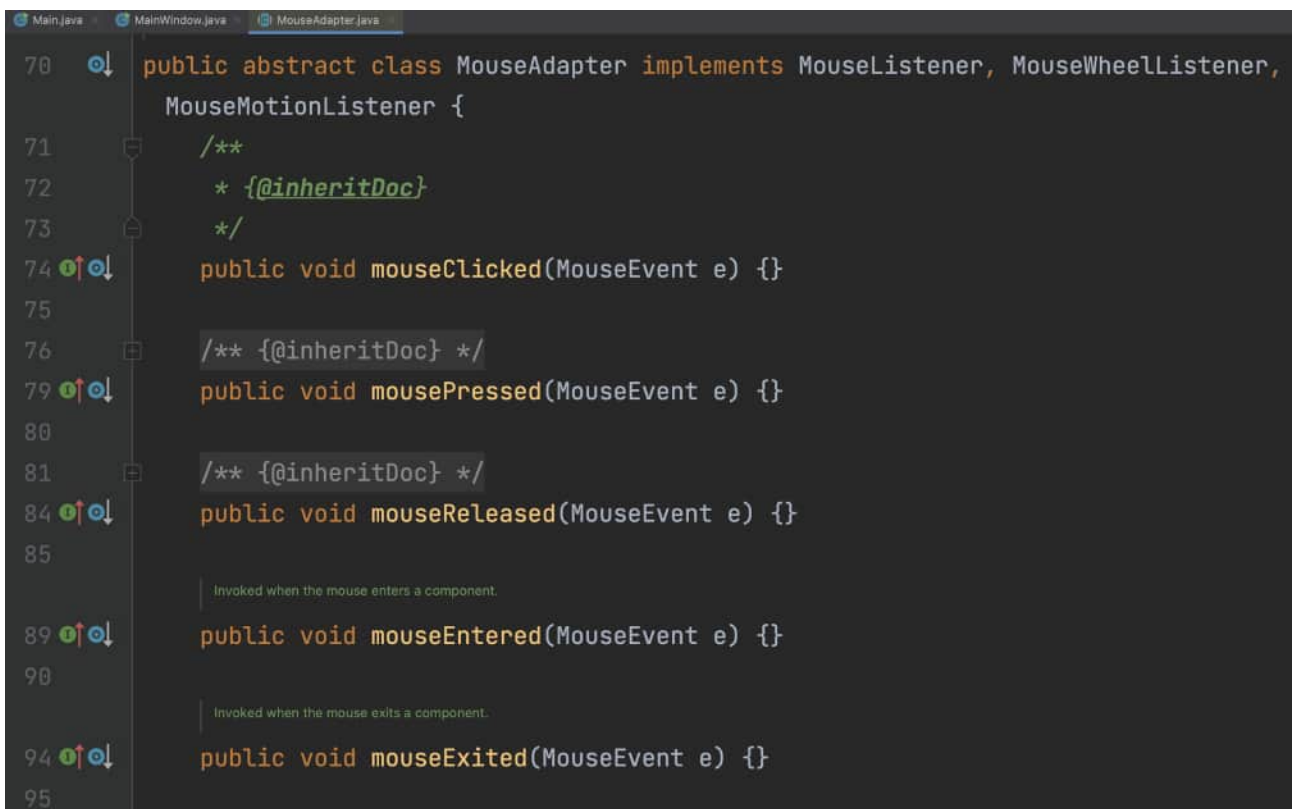
```

24     }
25     MainCanvas canvas = new MainCanvas(this);
26     canvas.addMouseListener(new Mouse);
27     add(canvas);
28     setVisible(true);
29 }

```

Рис. 7: Метод добавления слушателя мышки

Если в аргументе этого метода начать писать `new`, среда разработки предложит реализовать интерфейс `MouseListener`, но также предложит ещё один вариант – `MouseAdapter`. В исходниках класса `MouseAdapter` видно, что это класс, реализующий несколько интерфейсов, но только формально, то есть все реализации пустые.



```

70 public abstract class MouseAdapter implements MouseListener, MouseWheelListener,
71     MouseMotionListener {
72     /**
73      * {@inheritDoc}
74      */
75     public void mouseClicked(MouseEvent e) {}
76
77     /** {@inheritDoc} */
78     public void mousePressed(MouseEvent e) {}
79
80     /** {@inheritDoc} */
81     public void mouseReleased(MouseEvent e) {}
82
83     /**
84      * Invoked when the mouse enters a component.
85      */
86     public void mouseEntered(MouseEvent e) {}
87
88     /**
89      * Invoked when the mouse exits a component.
90      */
91     public void mouseExited(MouseEvent e) {}
92 }

```

Рис. 8: Содержимое класса MouseAdapter



Если попытаться это корректно перевести на русский язык, должно получиться: создай новый экземпляр анонимного класса, который наследуется от класса `MouseAdapter`, реализующего нужный интерфейс и



переопредели этот конкретный метод. Остальные оставь пустыми, потому что остальные действия можно игнорировать.

Как избежать таких «многоэтажных» и многострочных конструкций и при этом получить понятный код? Реализовать интерфейс в том классе, в котором в данный момент пишется код, и переопределить все методы интерфейса. В требуемые методы – написать реализацию, а туда, где требуется объект, реализующий интерфейс – передать ссылку this.

```

1 public class MainWindow extends JFrame implements
2     CanvasRepaintListener, MouseListener {
3     MainWindow() {
4         // ...
5         canvas.addMouseListener(this);
6     }
7     @Override public void mouseClicked(MouseEvent e) { }
8     @Override public void mousePressed(MouseEvent e) { }
9
10    @Override public void mouseReleased(MouseEvent e) {
11        System.out.println("Clicked!");
12    }
13    @Override public void mouseEntered(MouseEvent e) { }
14    @Override public void mouseExited(MouseEvent e) { }
15 }

```

Листинг 27: Пример реализации интерфейса «собой»

### 7.5.3. SOLID

Буква	Аббревиатура	Пояснение
S	SRP	Принцип единственной ответственности (single responsibility principle). Для каждого класса должно быть определено единственное назначение. Все ресурсы, необходимые для его осуществления, должны быть инкапсулированы в этот класс и подчинены только этой задаче.
O	OCP	Принцип открытости/закрытости (open-closed principle). «Программные сущности ... должны быть открыты для расширения, но закрыты для модификации».
L	LSP	Принцип подстановки Лисков (Liskov substitution principle). «Функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа не зная об этом».
I	ISP	Принцип разделения интерфейса (interface segregation principle). «Много интерфейсов, специально предназначенных для клиентов, лучше, чем один интерфейс общего назначения»
D	DIP	Принцип инверсии зависимостей (dependency inversion principle). «Зависимость на Абстракциях. Нет зависимости на что-то конкретное»

## 7.5.4. Вопросы для самопроверки

1. Программный интерфейс — это способ
  - (a) рисования объектов;
  - (b) взаимодействия объектов;
  - (c) взаимодействия программы с пользователем.
2. Анонимный класс — это класс без
  - (a) интерфейса;
  - (b) объекта;
  - (c) имени.
3. Поле в интерфейсе
  - (a) невозможно;
  - (b) `public static final`;
  - (c) `private final`.
4. Метод по-умолчанию
  - (a) можно переопределять;
  - (b) можно не переопределять;
  - (c) можно использовать с полем интерфейса;
  - (d) все варианты верны.

## 7.6. Исключения в графических интерфейсах пользователя

Поскольку графический интерфейс пользователя - это всегда многопоточность, и привычного терминала под рукой чаще всего нет, то возникают особенности обработки исключений. Как ловить? Как показывать?

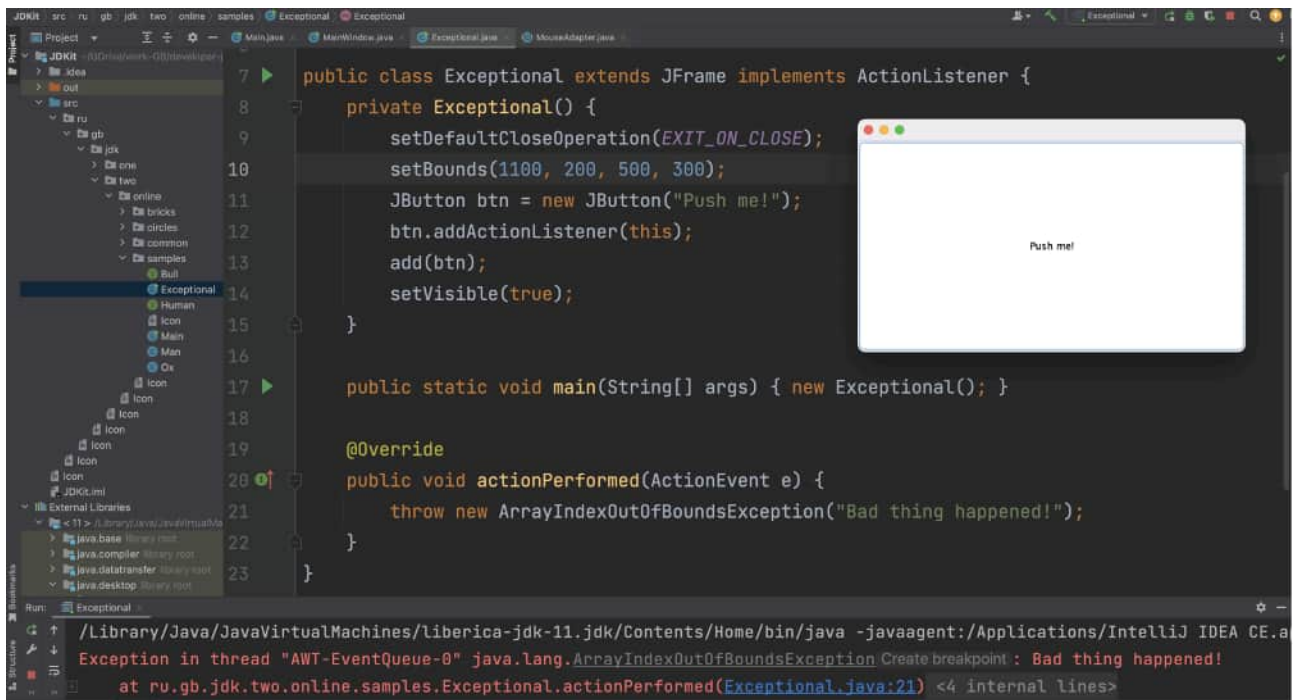


Рис. 10: Исключение, которое никогда не увидят

Например, есть окно, на котором есть кнопка. У кнопки есть обработчик, в котором что-то идёт не по плану, например, выход за пределы массива при подсчёте. Достаточно типичная ситуация. Возникает законный вопрос - как такое исключение поймать?

```

1 package ru.gb.jdk.two.online.samples;
2
3 import javax.swing.*;
4 import java.awt.event.ActionEvent;
5 import java.awt.event.ActionListener;
6
7 public class Exceptional extends JFrame implements ActionListener {
8     private Exceptional() {
9         setDefaultCloseOperation(EXIT_ON_CLOSE);
10        setBounds(1100, 200, 500, 300);
11        JButton btn = new JButton("Push me!");
12        btn.addActionListener(this);
13        add(btn);
14        setVisible(true);
15    }
16
17    public static void main(String[] args) { new Exceptional(); }
18
19    @Override
20    public void actionPerformed(ActionEvent e) {
21        throw new ArrayIndexOutOfBoundsException("Bad thing happened!");
22    }
23 }

```

Листинг 28: Пример окна с исключением

Если бы такое же исключение было выброшено в консольном приложении, программа бы завершилась, здесь же видно, что, несмотря на исключение в консоли, окно всё ещё открыто и приложение работает. Ясно видно, что исключения не завершают приложение. Во-первых, следует применять правильный способ создания главных окон в Swing, эта конструкция уже не должна быть магической: у класса `SwingUtilities` есть статический метод, в который передаётся экземпляр анонимного класса, реализующего интерфейс, и в переопределяемом методе создаётся окно.

```

1 public static void main(String[] args) {
2     SwingUtilities.invokeLater(new Runnable() {
3
4         @Override
5         public void run() {
6             new Exceptional();
7         }
8     });
9 }

```

Листинг 29: Верный способ создавать окна в Swing

Исключение происходит в специальном потоке EDT – Event Dispatching Thread. Этот поток совершает диспетчеризацию всех событий, происходящих во фреймворке Swing и является генератором других потоков. Метод из листинга 29 явно создаёт `JFrame` именно под управлением EDT. Если внимательно изучить текст исключения внизу экрана на рисунке 10, очевидно, что исключение возникло в потоке с названием `AWT-EventQueue-0`. Наличие у потока номера говорит о том, что таких очередей событий у приложения может быть много, и в каких-то из них могут возникать исключения. Исключение происходит в потоке. Обработчик исключений тоже содержится в потоке и называется `Thread.UncaughtExceptionHandler` и является интерфейсом. Интерфейс содержит один метод – непойманное исключение, который принимает на вход поток, в котором произошло исключение и объект исключения, которое произошло.

```

1 public class Exceptional extends JFrame implements
2     ActionListener, Thread.UncaughtExceptionHandler {
3
4     @Override
5     public void uncaughtException(Thread t, Throwable e) {
6
7     }
8 }

```

Листинг 30: Пример окна с исключением

Такие обработчики уже написаны и встроены в среду исполнения Java, но они только пишут в консоль стектрейс. Переопределив метод интерфейса в приложении – появляется возможность реагировать на исключения более сложно.

```

1 private Exceptional() {
2     Thread.setDefaultUncaughtExceptionHandler(this);
3     //...
4 }
5
6 @Override
7 public void uncaughtException(Thread t, Throwable e) {
8     JOptionPane.showMessageDialog(
9         null, e.getMessage(),
10        "Exception!", JOptionPane.ERROR_MESSAGE);
11 }

```

Листинг 31: Пример окна с исключением

В конструкторе (листинг 31), на строке 2 для потока устанавливается обработчик исключений по умолчанию, передаётся собственный объект окна. В самой функции обработки (строка 6), например, выводится на экран модальное окно с текстом исключения. Запустив приложение видно, что в консоли среды разработки нет сообщений об исключениях. Благодаря программным интерфейсам.

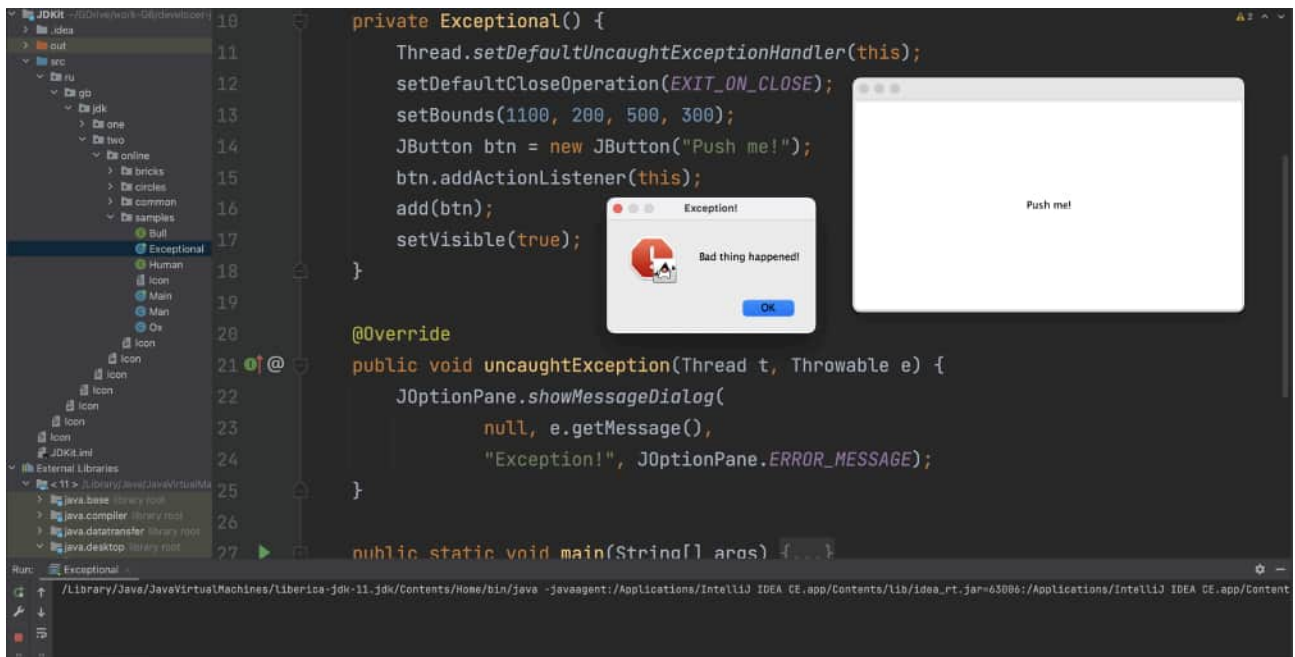


Рис. 11: Результат обработки исключения

## Практическое задание

1. Полностью разобраться с кодом.
2. Для приложения с шариками описать появление и убирание шариков по клику мышки левой и правой кнопкой соответственно.
3. Написать, выбросить и обработать такое исключение, которое не позволит создавать более, чем 15 шариков.
4. \*\* Написать ещё одно приложение, в котором на белом фоне будут перемещаться изображения формата png, лежащие в папке проекта.

## Термины, определения и сокращения

**implements** ключевое слово языка, указывающее на то, какой интерфейс (или интерфейсы) реализует класс.

**SOLID** Мнемонический акроним для первых пяти принципов, названных Робертом Мартином, которые означали пять основных принципов объектноориентированных проектирования и программирования.

**Адаптер** структурный шаблон проектирования, предназначенный для организации использования функций объекта, недоступного для модификации, через специально созданный интерфейс. Другими словами — это структурный паттерн проектирования, который позволяет объектам с несовместимыми интерфейсами работать вместе.

**Анон. класс** это класс, не имеющий имени и его создание происходит в момент инициализации объекта.

**Интерфейс** (API, англ. application programming interface) — описание способов взаимодействия одной компьютерной программы с другими (API приложения) или компонентов одной программы между собой (API объектов).

**Рендеринг** термин в компьютерной графике, обозначающий процесс получения изображения по модели с помощью компьютерной программы.

**Реализация** это класс, содержащий переопределённые методы интерфейса, выполняющий конкретные действия при вызове методов интерфейса.