ТЕМА 6
Многопоточное программирование

Цель лабораторной работы

Научиться создавать приложения, содержащие несколько потоков, и синхронизировать взаимодействие потоков между собой.

1 Создание потоков в приложениях на Java

Имеется два способа создать класс, экземплярами которого будут потоки выполнения: унаследовать класс от `java.lang.Thread` либо реализовать интерфейс `java.lang.Runnable`.

Интерфейс `Runnable` имеет декларацию единственного метода `public void run()`, который обеспечивает последовательность действий при работе потока.

Класс `Thread` уже реализует интерфейс `Runnable`, но с пустой реализацией метода `run()`, поэтому при создании экземпляра `Thread` автоматически создаётся (но не запускается!!!) поток. В потомках метод `run()` необходимо переопределять, поместив в него реализацию алгоритмов, которые должны выполняться в данном потоке. После выполнения метода `run()` поток прекращает существование.

1.1 Создание потока на основе класса Thread

Объект-поток создаётся с помощью конструктора. Имеется несколько перегруженных вариантов конструкторов, самый простой из них – с пустым списком параметров. Например, в классе `Thread` их заголовки выглядят так:
`public Thread()` – конструктор по умолчанию. Поток получает имя *system*.
`public Thread(String name)` – поток получает имя *name*.

Создание и запуск потока осуществляется следующим образом:

```
public class T1 extends Thread {
  public void run() {
    // Реализация метода run
  }
  // Другие методы класса
}
Thread thread1 = new T1();
thread1.start();
```

1.2 Создание потока на основе интерфейса Runnable

```
public class R1 implements Runnable {
  public void run() {
    // Реализация метода run
  }
  // Другие методы класса
}
```

```
Runnable myRunnable = new R1();
Thread thread2 = new Thread(myRunnable);
thread2.start();
```

Подобный способ позволяет создавать потоки на основе классов, уже включённых в иерархию наследования (т.е. тех, которые не могут выбрать `Thread` в качестве своего предка). При этом, однако, затрудняется доступ ко внутренним полям потока, т.к. формально их внутри нашего класса нет. Эту особенность приходится исправлять добавлением во внутреннее поле данных экземпляра класса ссылки на объект потока, в котором он выполняется.

```
public class R1 implements Runnable {
  Thread containerThread;
  public void run() {
    // Реализация метода run
  }
  public void setThread(Thread containerThread) {
    this.containerThread = containerThread;
  }
  // Другие методы класса
}
Runnable myRunnable = new R1();
Thread thread2 = new Thread(myRunnable);
myRunnable.setThread(thread2);
thread2.start();
```

С помощью ссылки на объект потока можно обращаться к его полям данных и методам, например, считывая текущий приоритет потока:

```
containerThread.getPriority();
```

1.3 Синхронизация потоков

Если разные потоки получают доступ к одним и тем же данным, причём один из них или они оба меняют эти данные, для них требуется обеспечить установить разграничение доступа. Пока один поток меняет данные, второй не должен иметь права их читать или менять, а должен дожидаться окончания доступа к данным первого потока. В Java для этих целей служит оператор `synchronize`. Такой тип синхронизации называется **синхронизацией на ресурсах** и обеспечивает блокировку данных на то время, которое необходимо потоку для выполнения тех или иных действий. Блокировка такого рода осуществляется на основе концепции мониторов и заключается в следующем:

Под монитором понимается некая управляющая конструкция, обеспечивающая монопольный доступ к объекту. Если во время выполнения синхронизованного метода объекта другой поток попробует обратиться к методам или данным этого объекта, он будет блокирован до тех пор, пока не закончится выполнение синхронизованного метода. При запуске

синхронизованного метода говорят, что **объект входит в монитор**, при завершении – что **объект выходит из монитора**. При этом поток, внутри которого вызван синхронизованный метод, считается владельцем данного монитора.

Имеется два способа синхронизации на ресурсах: синхронизация на объектах и синхронизация на методах.

1.3.1 Синхронизация на объектах

Синхронизация на объекте `obj` осуществляется следующим образом:

```
synchronized(obj) {
  method1(obj);
  obj.method2();
  // Другие синхронизированные действия
}
```

В данном случае в качестве синхронизованного оператора выступает участок кода в фигурных скобках. Во время выполнения этого участка доступ к объекту `obj` блокируется для всех других потоков, т.е. пока будет выполняться вызов оператора, выполнение вызова любого синхронизованного метода или синхронизованного оператора для этого объекта будет приостановлено до окончания работы оператора. Данный способ синхронизации обычно используется для экземпляров классов, разработанных без расчёта на работу в режиме многопоточности.

1.3.2 Синхронизация на методах

Второй способ синхронизации на ресурсах используется при разработке классов, рассчитанных на взаимодействия в многопоточной среде. При этом методы, критичные к атомарности операций с данными (обычно – требующие согласованное изменение нескольких полей данных), объявляются как синхронизованные с помощью модификатора `synchronized`:

```
public class MyClass {
  public synchronized void myMethod() {
    // Реализация метода
  }
}
```

Вызов `obj.myMethod()` приведёт к вхождению объекта в монитор и пока он будет выполняться, доступ из других потоков к объекту будет блокирован – выполнение любого синхронизованного метода или оператора для этого объекта будет приостановлено до окончания работы метода.

Если синхронизованный метод является не методом объекта, а методом класса, при вызове метода в монитор входит класс, и приостановка до окончания работы метода будет относиться ко всем вызовам синхронизованных методов данного класса.

1.3.3 Синхронизация на событиях

Кроме синхронизации на данных имеется **синхронизация на событиях**, когда параллельно выполняющиеся потоки приостанавливаются вплоть до наступления некоторого события, о котором им сигнализирует другой поток. Основными операциями при таком типе синхронизации являются **ждать** и **оповестить.**

В Java синхронизацию по событиям обеспечивают методы, заданные в классе `Object` и наследуемые всеми остальными классами:

- `wait()` – поток, внутри которого какой-либо объект вызвал данный метод, приостанавливает работу своего метода `run()` вплоть до поступления уведомления (посредством `notify()` или `notifyAll()`**)** объекту, вызвавшему остановку. При неправильной попытке “разбудить” поток соответствующий код компилируется, но при запуске инициирует исключение `llegalMonitorStateException`.
- `notify()` – оповещение, приводящее к возобновлению работы потока, ожидающего выхода данного объекта из монитора. Если таких потоков несколько, выбирается один из них (какой именно – зависит от реализации JVM).
- `notifyAll()` – оповещение, приводящее к возобновлению работы всех потоков,ожидающих выхода данного объекта из монитора.

Метод `wait()` для любого объекта следует использовать следующим образом – организовать цикл `while`, в котором следует выполнять оператор `wait()`:

```
synchronized(obj) {
  while(!<condition>)
    obj.wait();
    // Выполнение операторов после разблокирования
}
```

Не следует беспокоиться, что постоянно работая цикл `while` будет занимать ресурсы процессора. После вызова `obj.wait()` поток, в котором находится указанный код, засыпает и перестаёт работать. На время сна метод `wait()` снимает блокировку с объекта `obj`, задаваемую оператором `synchronized(obj)`, что позволяет другим потокам обращаться к объекту с вызовом `obj.notify()` или `obj.notifyAll()`.

Альтернативным способом реализации синхронизации на событиях является замена используемой в предыдущем фрагменте кода синхронизации на объекте на синхронизацию на его методах. В этом случае мы можем сконструировать специальный объект, контролирующий и ограничивающий одновременный доступ потоков:

```
class Lock {
  private boolean locked;
  public synchronized void checkIfLocked() throws InterrupedException {
    while(locked) wait();
  }
  public synchronized void lock() {
    locked = true;
  }
  public synchronized void unlock() {
    locked = false;
    notifyAll();
  }
}
```

2 Краткая справка по необходимым программным компонентам

Важные методы экземпляров класса `Thread`

Имя константы или метода	Описание
`void run()`	Метод, который обеспечивает последовательность действий во время жизни потока.
`void start()`	Вызывает выполнение текущего потока, в том числе запуск его метода `run()` в нужном контексте. Может быть вызван всего один раз.
`void yield()`	Вызывает временную приостановку потока, с передачей управления другим потокам.
`long getId()`	Возвращает уникальный (только во время жизни) идентификатор потока
`String getName()`	Возвращает имя потока, которое ему было задано при создании или методом `setName()`
`void setName(String name)`	Устанавливает новое имя потока.
`int getPriority()`	Возвращает приоритет потока.
`void setPriority(int newPriority)`	Устанавливает приоритет потока.
`String toString()`	Возвращает строковое представление объекта потока, в том числе – его имя, группу, приоритет.
`void sleep(long millis)`	Вызывает приостановку (засыпание) потока на `millis` миллисекунд. При этом все блокировки потока сохраняются.
`void interrupt()`	Прерывает сон потока, вызванный вызовами `wait()` или `sleep()`, устанавливая ему статус прерванного. При этом возбуждается проверяемая исключительная ситуация

	`InterruptedException`.
`boolean isInterrupted()`	Возвращает текущее состояние статуса прерывания потока без изменения значения статуса.
`void join()`	Переводит поток в режим умирания – ожидания завершения (смерти). Обычно используется для завершения главным потоком работы всех дочерних пользовательских потоков («слияния» их с главным потоком).
`boolean isAlive()`	Возвращает true в случае, когда текущий поток жив (не умер). Отметим, что даже если поток завершился, от него остаётся объект-«призрак», отвечающий на запрос `isAlive()` значением `false`.

3 Пример приложения

Задание: составить программу анимации мячей, летающих в пространстве окна. При столкновении мяча с краем окна происходит его «отскок» в обратную сторону, т.е. нормальная составляющая вектора скорости изменяет свой знак, а тангенциальная остаётся без изменений. Мячи раскрашены в разные цвета, имеют различный размер (от 3 до 40 точек) и различную скорость. Величина размера мячей случайна, равно распределена на отрезке от `MIN_RADIUS` до `MAX_RADIUS`. Скорость полёта мяча определяется исходя из его радиуса по формуле $5 \cdot MAX\_RADIUS/radius$, где *radius* – радиус мяча. Начальное положение мяча и направление вектора его скорости случайны. Добавление новых мячей происходит через элемент меню «*Мячи* → *Добавить мяч*». Предусмотрена возможность приостановки движения всех мячей с помощью элемента меню «*Управление* → *Приостановить движение*», а также его возобновления с помощью «*Управление* → *Возобновить движение*». Внешний вид окна представлен на рисунке 3.1.

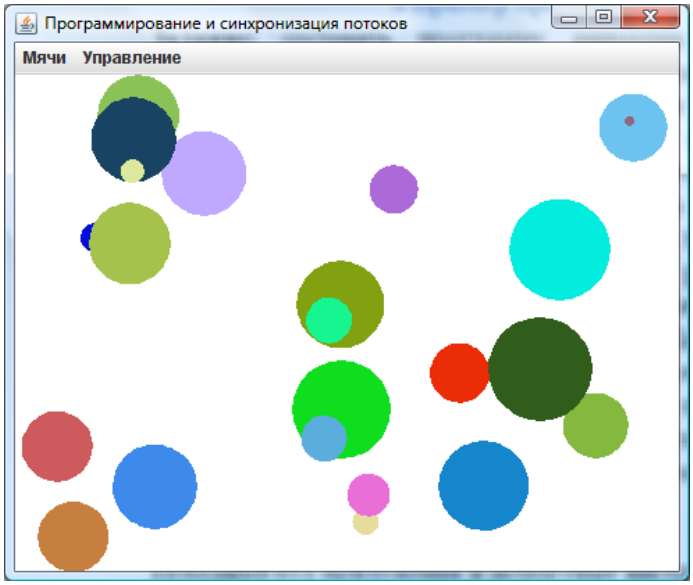

Рисунок 3.1 – Внешний вид главного окна приложения

3.1 Структура приложения

В структуре приложения можно выделить следующие элементы:

Рабочее поле, хранящее список всех летающих мячей, ответственное за регулярную перерисовку области окна (с помощью таймера) и управление их движением.

Отскакивающий мяч, ответственный за постоянный перерасчёт своих координат на основе скорости и направления движения, а также за своё отображение в рамках заданного контекста устройства.

Главное окно приложения, организующее рабочее пространство окна, показ главного меню, реакцию на действия пользователя.

Диаграмма классов приложения, на которой видны выделенные элементы, приведена на рисунке 3.2:

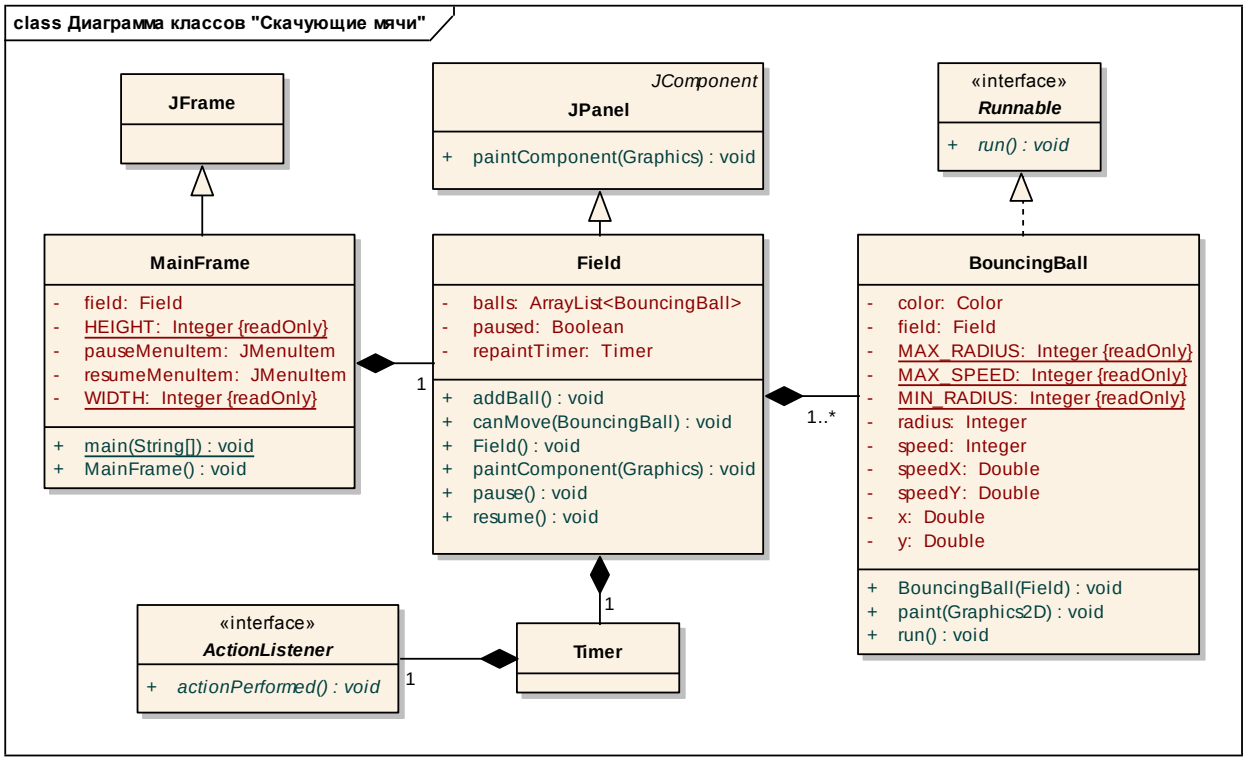

Рисунок 3.2 – Диаграмма классов приложения

3.2 Подготовительный этап

Этап предполагает создание каркаса приложения (см. шаги 3.1 – 3.4 лабораторной работы №1). Назовём главный класс приложения `MainFrame`. В роли класса-предка для него выступает класс `JFrame`.

3.3 Реализация класса рабочего поля Field

Класс рабочего поля `Field` является наследником класса панели `JPanel` и выступает в роли контейнера для перемещающихся в рамках поля мячей.

Дополнительно к этому, класс `Field` представляет собой объект-семафор, являющийся точкой синхронизации для всех потоков.

Для хранения списка всех мячей в классе `Field` как поле данных объектов класса присутствует динамический список:

```
// Динамический список скачущих мячей
private ArrayList<BouncingBall> balls = new ArrayList<BouncingBall>(10);
```

Текущее состояние запрета на движение (или разрешения) хранится в булевой переменной-флаге `paused`:

```
// Флаг приостановленности движения
private boolean paused;
```

Для обеспечения периодической перерисовки области поля необходимо воспользоваться классом `Timer`, в задачи которого входит генерация экземпляров события типа `ActionEvent` через заданные промежутки времени. Получателем и обработчиком генерируемых событий сделаем анонимный класс, реализующий интерфейс `ActionListener`, единственной задачей которого является инициирование перерисовки окна:

```
// Класс таймер отвечает за регулярную генерацию события типа ActionEvent
// При создании его экземпляра используется анонимный класс,
// реализующий интерфейс ActionListener
private Timer repaintTimer = new Timer(10, new ActionListener() {
      public void actionPerformed(ActionEvent ev) {
            // Задача обработчика события ActionEvent одна - перерисовка окна
            repaint();
      }
});
```

Так как при создании нового объекта класса `Field` никаких сложных действий не требуется, то и его конструктор является простым:

```
// Конструктор класса BouncingBall
public Field() {
      // Установить цвет заднего фона белым
      setBackground(Color.WHITE);
      // Запустить таймер перерисовки области
      repaintTimer.start();
}
```

Добавление на поле нового мяча осуществляется простой вставкой в конец списка мячей нового экземпляра класса `BouncingBall`, так как все дополнительные действия, связанные с определением начальной позиции, радиуса мяча, его цвета и вектора скорости экземпляр мяча выполняет самостоятельно в конструкторе:

```
// Метод добавления нового мяча в список
```

```
public void addBall() {
	// Заключается в добавлении в список нового экземпляра BouncingBall
	// Всю инициализацию BouncingBall выполняет сам в конструкторе
	balls.add(new BouncingBall(this));
}
```

Задачу отображения мячей на экране возложим на сами объекты мячей, с учётом чего код метода `paintComponent()` примет следующий вид:

```
// Унаследованный от JPanel метод перерисовки компонента
public void paintComponent(Graphics g) {
	// Вызвать версию метода, унаследованную от предка
	super.paintComponent(g);
	Graphics2D canvas = (Graphics2D) g;
	// Последовательно запросить прорисовку от всех мячей, хранимых в списке
	for (BouncingBall ball: balls) {
		ball.paint(canvas);
	}
}
```

Для обеспечения синхронизации движения всех мячей (т.е. синхронизации потоков) предназначены три метода: `pause()`, `resume()` и `canMove()`. Все они являются синхронизированными (объявлены с ключевым словом `synchronized`), что ограничивает число потоков, одновременно выполняющихся внутри данных методов, одним.

Наиболее простым является метод `pause()`, т.к. всё, что ему нужно, это воспользовавшись эксклюзивным доступом к переменной `pause` (она изменяется только внутри синхронизированных методов) установить её значение в `true`:

```
// Синхронизированный, т.е. только 1 поток может одновременно быть внутри
public synchronized void pause() {
	// Включить режим паузы
	paused = true;
}
```

Метод `canMove()` последовательно вызывается каждым из мячей перед выполнением операции перерасчёта координат, поэтому является удобным местом для контролируемой приостановки работы приложения. Так как метод синхронизированный, то выполняться он будет всеми потоками не одновременно, а строго последовательно (фактически, потоки выстроятся в очередь на выполнение метода). В теле метода изучается значение флаговой переменной `paused`, и если оно истинно (т.е. включен режим паузы), то посредством метода `wait()` проводится «усыпление» потока, и право начать выполнение метода `canMove()` передаётся следующему потоку из очереди. Если за этот временной интервал значение флага `paused` не изменилось, то и следующий поток заснёт, а запущен будет очередной по цепочке. Итоговым результатом станет то, что за короткое время после установки флага `paused` в

`true` все потоки обратившись к методу `canMove()` перейдут в спящее состояние:

```
// Синхронизированный метод проверки, может ли мяч двигаться
// (не включен ли режим паузы?)
public synchronized void canMove(BouncingBall ball)
                                          throws InterruptedException {
      if (paused) {
            // Если режим паузы включен, то поток, зашедший внутрь
            // данного метода, засыпает
            wait();
      }
}
```

Для снятия паузы и пробуждения всех спящих потоков предназначен метод `resume()`, который изменяет состояние флаговой переменной `paused` на `false`, после чего с помощью метода `notifyAll()` осуществляет пробуждение всех заснувших потоков:

```
// Синхронизированный, т.е. только 1 поток может одновременно быть внутри
public synchronized void resume() {
      // Выключить режим паузы
      paused = false;
      // Будим все ожидающие продолжения потоки
      notifyAll();
}
```

3.4 Реализация класса прыгающего мяча BouncingBall

Задачей класса мяча является инициализация внутренних атрибутов объекта в момент его создания (в теле конструктора), выполнение операций по визуализации мяча на экране с учётом текущих значений атрибутов (координаты, размер, цвет), а также постоянный пересчёт координат мяча исходя из направления его движения, скорости и размеров поля.

Такие атрибуты мяча, как его размер, направление движения, начальные координаты и цвет являются произвольными, и генерируются в конструкторе случайным образом. Скорость зависит от размера мяча и определяет промежуток времени, на который будет приостанавливаться выполнение потока, после каждого перерасчёта координат:

```
// Конструктор класса BouncingBall
public BouncingBall(Field field) {
      // Необходимо иметь ссылку на поле, по которому прыгает мяч,
      // чтобы отслеживать выход за его пределы через getWidth(), getHeight()
      this.field = field;
      // Радиус мяча случайного размера
      radius = new Double(Math.random()*(MAX_RADIUS - MIN_RADIUS)).intValue()
+ MIN_RADIUS;
      // Скорость зависит от диаметра мяча, чем он больше, тем медленнее
      speed = new Double(Math.round(5*MAX_SPEED / radius)).intValue();
      if (speed>MAX_SPEED) {
            speed = MAX_SPEED;
```

```
        }
        // Начальное направление случайно, угол в пределах от 0 до 2PI
        double angle = Math.random()*2*Math.PI;
        // Вычисляются горизонтальная и вертикальная компоненты скорости
        speedX = 3*Math.cos(angle);
        speedY = 3*Math.sin(angle);
        // Цвет мяча выбирается случайно
        color = new Color((float)Math.random(), (float)Math.random(),
                          (float)Math.random());
        // Начальное положение мяча случайно
        x = Math.random()*(field.getSize().getWidth() - 2*radius) + radius;
        y = Math.random()*(field.getSize().getHeight() - 2*radius) + radius;
        // Создаём новый экземпляр потока, передавая аргументом ссылку на класс,
        // реализующий Runnable (т.е. на себя)
        Thread thisThread = new Thread(this);
        // Запускаем поток
        thisThread.start();
}
```

Метод `paint()`, выполняющий отображение мяча на экране, содержит ряд простейших обращений к Java 2D API для рисования закрашенного эллипса:

```
// Метод прорисовки самого себя
public void paint(Graphics2D canvas) {
        canvas.setColor(color);
        canvas.setPaint(color);
        Ellipse2D.Double ball = new Ellipse2D.Double(x - radius, y - radius,
                                                     2*radius, 2*radius);
        canvas.draw(ball);
        canvas.fill(ball);
}
```

Метод `run()`, выполняемый для каждого мяча в отдельном потоке, содержит основную логику по периодическому перерасчёту положения мяча:

```
// Метод run() исполняется внутри потока.
// Когда он завершает работу, то завершается и поток
public void run() {
        try {
                // Крутим бесконечный цикл, т.е. пока нас не прервут,
                // мы не намерены завершаться
                while (true) {
                        // Синхронизация потоков делается на самом объекте поля
                        // Если движение разрешено - управление будет возвращено
                        // В противном случае - активный поток заснёт
                        field.canMove(this);
                        if (x + speedX <= radius) {
                                // Достигли левой стенки, отскакиваем право
                                speedX = -speedX;
                                x = radius;
                        } else
                        if (x + speedX >= field.getWidth() - radius) {
                                // Достигли правой стенки, отскок влево
                                speedX = -speedX;
                                x = new Double(field.getWidth() - radius).intValue();
                        } else
                        if (y + speedY <= radius) {
```

```
                        // Достигли верхней стенки
                        speedY = -speedY;
                        y = radius;
                    } else
                    if (y + speedY >= field.getHeight() - radius) {
                        // Достигли нижней стенки
                        speedY = -speedY;
                        y = new Double(field.getHeight() - radius).intValue();
                    } else {
                        // Просто смещаемся
                        x += speedX;
                        y += speedY;
                    }
                    // Засыпаем на X миллисекунд, где X определяется исходя из
                    // скорости: Скорость = 1 (медленно), засыпаем на 15мс.
                    // Скорость = 15 (быстро), засыпаем на 1 мс.
                    Thread.sleep(16-speed);
                }
            } catch (InterruptedException ex) {
                // Если нас прервали, то просто выходим (завершаемся)
            }
}
```

3.5 Реализация главного класса приложения MainFrame

Задачами главного класса приложения является создание и компоновка элементов пользовательского интерфейса, а также конструирование и показ главного окна приложения. Эти задачи решаются способом, аналогичным рассмотренным в лабораторных работах 3 – 4, и подробно рассматриваться не будут. С полным исходным кодом класса можно ознакомиться в Приложении 1.

4 Задания

4.1 Вариант А

Модифицировать (по вариантам) приложение прыгающих мячей.

№ п/п	Цель модификаций
1	Добавить возможность выборочной остановки мячей малого радиуса (до 10 точек).
2	Добавить возможность выборочной остановки мячей, двигающихся с малой скоростью (время сна потока более 8 мс).
3	Добавить возможность выборочной остановки мячей, угол скорости которых находится между 90 и 180 градусов (2-я четверть).
4	Добавить возможность выборочной остановки зелёных мячей (у которых интенсивность зелёной компоненты более чем в два раза превышает сумму двух других компонент).
5	Добавить возможность выборочной остановки мячей большого радиуса (более 30 точек).
6	Добавить возможность выборочной остановки мячей, двигающихся с большой скоростью (время сна потока менее 8 мс).
7	Добавить возможность выборочной остановки мячей, угол скорости которых находится между 0 и 90 градусов (1-я четверть).
8	Добавить возможность выборочной остановки синих мячей (у которых интенсивность синей компоненты более чем в два раза превышает сумму двух других компонент).
9	Ограничить количество останавливаемых мячей значением 5 (другие не останавливаются, а продолжают движение).
10	Добавить возможность выборочной остановки мячей, угол скорости которых находится между 180 и 270 градусов (3-я четверть).
11	Добавить возможность выборочной остановки красных мячей (у которых интенсивность красной компоненты более чем в два раза превышает сумму двух других компонент).
12	Добавить возможность выборочной остановки мячей, угол скорости которых находится между 270 и 360 градусов (4-я четверть).

4.2 Вариант В

Модифицировать (по вариантам) приложение прыгающих мячей.

№ п/п	Цель модификаций
1	Добавить в приложение эффект «трение», в активированном состоянии постоянно снижающей скорость движения мячей. Коэффициент силы трения вводить через меню.
2	Добавить в приложение эффект «замедленная съёмка», позволяющий изменять скорость течения времени (кнопки «+» и «–», позволяющие управлять скоростью передвижения всех мячей сразу). При нажатии кнопки «+» скорость возрастает, при нажатии «–» – уменьшается.
3	Добавить в приложение эффект «магнетизм» (через флажок в меню). В случае, если эффект включен, мячи дотронувшись до стенок «прилипают» к ним. Когда эффект выключается, все прилипшие мячи разом отскакивают от стенок и продолжают движение с теми направлениями и скоростями, которые были до «прилипания».

4	Сняв ограничение на минимальный размер мяча, добавить в приложение эффект «наждачная бумага» (через флажок в меню). Если режим включен, то при каждом ударе мяча о стенку его радиус уменьшается на X (задаётся через меню). Если радиус мяча достиг нуля, то мяч уничтожается.
5	Сняв ограничение на максимальный размер мяча, добавить в приложение эффект «снежный ком» (через флажок в меню). Если режим включен, то за каждые X точек пройденного пути радиус мяча увеличивается на Y (X и Y задаются через меню). Если скорость мяча уменьшилась до нуля, то его координаты перестают пересчитываться.
6	При отображении мяча на экране выводить рядом строковый идентификатор мяча (он может быть одинаковым у нескольких мячей) и добавить в приложение эффект «мы-команда». Для активации эффекта необходимо запросить у пользователя (через диалоговое окно) строку и остановить все мячи, у которых идентификатор отличается от введенной строки.
7	Добавить в приложение эффект «харизма». При активации эффекта все мячи оставляют свои траектории движения и стремятся поместить свой центр в точку, над которой сейчас находится курсор. При изменении положения курсора мячи смещаются аналогичным образом. При выключении эффекта мячи разлетаются в случайные стороны.
8	Добавить в приложение эффект «обжора». При его активации на поле появляется «обжора» (большой мяч радиусом 100 точек), который подчиняется тем же законам, что и маленькие мячи (отскакивает от стенок). Но в том случае, если во время своего движения маленький мяч влетает внутрь обжоры, то выйти за его пределы он не может, каждый раз отскакивая от внутренних стенок. При деактивации эффекта обжора исчезает, а маленькие шары находящиеся в нём разлетаются во все стороны.
9	Отказавшись от начального распахивания окна приложения на весь экран, а сделав его размером 300x200 точек, реализовать эффект «теория относительности». В случае, когда эффект активирован, то летают не только мячи внутри окна, но и само окно перемещается по экрану, упруго отскакивая от его краёв.

4.3 Вариант C

Модифицировать (по вариантам) приложение прыгающих мячей.

№ п/п	Цель модификаций
1	Добавить возможность изменить направление и скорость движения мяча с помощью курсора мыши. Для этого необходимо щёлкнуть на мяче, затем не отпуская нажатой кнопки мыши сместить курсор в требуемом направлении, отпустить курсор мыши. По начальной (нажата кнопка мыши) и конечной (отпущена кнопка мыши) точкам определить новое направление движения мяча, по скорости (расстояние между точками, делённое на длительность) жеста – новую скорость движения мяча. На всё время, пока кнопка мыши удерживается в нажатом положении, все мячи должны приостановить своё движение.
2	Ограничив количество мячей восемью и считая, что они состоят из одинакового материала, реализовать модель столкновения твёрдых шаров (см. курс механики).
3	Добавить в приложение объект прямоугольной формы (препятствие), от которого мячи отскакивают так же, как и от стен. Связать перемещение препятствия внутри поля с перемещением курсора мыши.
4	Ограничить количество мячей одним и добавить возможность появления на поле разрушаемых препятствий (кирпичей). Каждый кирпич может выдержать N ударов мячом, после чего он разрушается и исчезает. Визуально прочность кирпича должна

	показываться цифрой на его поверхности. Столкновение мяча с кирпичом рассчитывается аналогично столкновению мяча и стенки. Подсказка: необходимо рассматривать всё поле как матрицу, при этом кирпичи могут стоять только в определённых позициях (ячейках), т.е. кирпич не может находиться на стыке двух ячеек.
5	Добавить на поле три типа объектов: «Конструктор», «Деструктор» и «Телепорт». Конструктор создаёт копию попавшего в него мяча, которая вылетает в случайном направлении. Деструктор уничтожает попавший в него мяч. Телепорт перемещает попавший в него мяч в целевую область (телепорт включает область-источник и область-приёмник). Для активации данных предметов необходимо, чтобы мяч целиком поместился в них (если он не поместился, а пролетел «по касательной», то активации не происходит).
6	Ограничить количество мячей одним и добавить в приложение два прямоугольных объекта (ракетки), двигающиеся вдоль противоположных сторон экрана. Мяч отскакивает либо от боковых стенок, либо от поверхности ракеток. При контакте мяча с верхней или нижней стенками противнику владельца стенки начисляется +1 очко. Одна ракетка управляется пользователем (с клавиатуры), вторая – программой. Допустимо перемещение ракеток только по горизонтали, но скорость их перемещения ограничена. Реализовать процедуру подсчёта очков, по достижении кем-либо значения 10 – объявить победителя.

Приложение 1. Исходный код приложения

Класс рабочего поля Field

```
package bsu.rfe.java.teacher;

import java.awt.Color;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.ArrayList;
import javax.swing.JPanel;
import javax.swing.Timer;

@SuppressWarnings("serial")
public class Field extends JPanel {

	// Флаг приостановленности движения
	private boolean paused;
	// Динамический список скачущих мячей
	private ArrayList<BouncingBall> balls = new ArrayList<BouncingBall>(10);

	// Класс таймер отвечает за регулярную генерацию событий ActionEvent
	// При создании его экземпляра используется анонимный класс,
	// реализующий интерфейс ActionListener
	private Timer repaintTimer = new Timer(10, new ActionListener() {
		public void actionPerformed(ActionEvent ev) {
			// Задача обработчика события ActionEvent - перерисовка окна
			repaint();
		}
	});

	// Конструктор класса BouncingBall
	public Field() {
		// Установить цвет заднего фона белым
		setBackground(Color.WHITE);
		// Запустить таймер
		repaintTimer.start();
	}

	// Унаследованный от JPanel метод перерисовки компонента
	public void paintComponent(Graphics g) {
		// Вызвать версию метода, унаследованную от предка
		super.paintComponent(g);
		Graphics2D canvas = (Graphics2D) g;
		// Последовательно запросить прорисовку от всех мячей из списка
		for (BouncingBall ball: balls) {
			ball.paint(canvas);
		}
	}

	// Метод добавления нового мяча в список
	public void addBall() {
		//Заключается в добавлении в список нового экземпляра BouncingBall
		// Всю инициализацию положения, скорости, размера, цвета
		// BouncingBall выполняет сам в конструкторе
		balls.add(new BouncingBall(this));
	}
```

```
	// Метод синхронизированный, т.е. только один поток может
	// одновременно быть внутри
	public synchronized void pause() {
		// Включить режим паузы
		paused = true;
	}

	// Метод синхронизированный, т.е. только один поток может
	// одновременно быть внутри
	public synchronized void resume() {
		// Выключить режим паузы
		paused = false;
		// Будим все ожидающие продолжения потоки
		notifyAll();
	}

	// Синхронизированный метод проверки, может ли мяч двигаться
	// (не включен ли режим паузы?)
	public synchronized void canMove(BouncingBall ball) throws
InterruptedException {
		if (paused) {
			// Если режим паузы включен, то поток, зашедший
			// внутрь данного метода, засыпает
			wait();
		}
	}
}
```

Класс прыгающего мяча BouncingBall

```
package bsu.rfe.java.teacher;

import java.awt.Color;
import java.awt.Graphics2D;
import java.awt.geom.Ellipse2D;

public class BouncingBall implements Runnable {

	// Максимальный радиус, который может иметь мяч
	private static final int MAX_RADIUS = 40;
	// Минимальный радиус, который может иметь мяч
	private static final int MIN_RADIUS = 3;
	// Максимальная скорость, с которой может летать мяч
	private static final int MAX_SPEED = 15;

	private Field field;
	private int radius;
	private Color color;

	// Текущие координаты мяча
	private double x;
	private double y;

	// Вертикальная и горизонтальная компонента скорости
	private int speed;
	private double speedX;
	private double speedY;

	// Конструктор класса BouncingBall
	public BouncingBall(Field field) {
		// Необходимо иметь ссылку на поле, по которому прыгает мяч,
		// чтобы отслеживать выход за его пределы
```

```
            // через getWidth(), getHeight()
            this.field = field;
            // Радиус мяча случайного размера
            radius = new Double(Math.random()*(MAX_RADIUS -
MIN_RADIUS)).intValue() + MIN_RADIUS;
            // Абсолютное значение скорости зависит от диаметра мяча,
            // чем он больше, тем медленнее
            speed = new Double(Math.round(5*MAX_SPEED / radius)).intValue();
            if (speed>MAX_SPEED) {
                  speed = MAX_SPEED;
            }
            // Начальное направление скорости тоже случайно,
            // угол в пределах от 0 до 2PI
            double angle = Math.random()*2*Math.PI;
            // Вычисляются горизонтальная и вертикальная компоненты скорости
            speedX = 3*Math.cos(angle);
            speedY = 3*Math.sin(angle);
            // Цвет мяча выбирается случайно
            color = new Color((float)Math.random(), (float)Math.random(),
(float)Math.random());
            // Начальное положение мяча случайно
            x = Math.random()*(field.getSize().getWidth()-2*radius) + radius;
            y = Math.random()*(field.getSize().getHeight()-2*radius) + radius;
            // Создаём новый экземпляр потока, передавая аргументом
            // ссылку на класс, реализующий Runnable (т.е. на себя)
            Thread thisThread = new Thread(this);
            // Запускаем поток
            thisThread.start();
      }

      // Метод run() исполняется внутри потока. Когда он завершает работу,
      // то завершается и поток
      public void run() {
            try {
                  // Крутим бесконечный цикл, т.е. пока нас не прервут,
                  // мы не намерены завершаться
                  while(true) {
                        // Синхронизация потоков на самом объекте поля
                        // Если движение разрешено - управление будет
                        // возвращено в метод
                        // В противном случае - активный поток заснёт
                        field.canMove(this);
                        if (x + speedX <= radius) {
                              // Достигли левой стенки, отскакиваем право
                              speedX = -speedX;
                              x = radius;
                        } else
                        if (x + speedX >= field.getWidth() - radius) {
                              // Достигли правой стенки, отскок влево
                             speedX = -speedX;
                             x=new Double(field.getWidth()-radius).intValue();
                        } else
                        if (y + speedY <= radius) {
                              // Достигли верхней стенки
                              speedY = -speedY;
                              y = radius;
                        } else
                        if (y + speedY >= field.getHeight() - radius) {
                              // Достигли нижней стенки
                            speedY = -speedY;
                            y=new Double(field.getHeight()-radius).intValue();
                        } else {
                              // Просто смещаемся
                              x += speedX;
```

```
                        y += speedY;
                    }
                    // Засыпаем на X миллисекунд, где X определяется
                    // исходя из скорости
                    // Скорость = 1 (медленно), засыпаем на 15 мс.
                    // Скорость = 15 (быстро), засыпаем на 1 мс.
                    Thread.sleep(16-speed);
                }
            } catch (InterruptedException ex) {
                // Если нас прервали, то ничего не делаем
                // и просто выходим (завершаемся)
            }
        }

        // Метод прорисовки самого себя
        public void paint(Graphics2D canvas) {
            canvas.setColor(color);
            canvas.setPaint(color);
            Ellipse2D.Double ball = new Ellipse2D.Double(x-radius, y-radius,
                                                         2*radius, 2*radius);
            canvas.draw(ball);
            canvas.fill(ball);
        }

}
```

Класс главного окна приложения MainFrame

```
package bsu.rfe.java.teacher;

import java.awt.BorderLayout;
import java.awt.Toolkit;
import java.awt.event.ActionEvent;
import javax.swing.AbstractAction;
import javax.swing.Action;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;

@SuppressWarnings("serial")
public class MainFrame extends JFrame {

        // Константы, задающие размер окна приложения, если оно
        // не распахнуто на весь экран
        private static final int WIDTH = 700;
        private static final int HEIGHT = 500;

        private JMenuItem pauseMenuItem;
        private JMenuItem resumeMenuItem;

        // Поле, по которому прыгают мячи
        private Field field = new Field();

        // Конструктор главного окна приложения
        public MainFrame() {
            super("Программирование и синхронизация потоков");
            setSize(WIDTH, HEIGHT);
            Toolkit kit = Toolkit.getDefaultToolkit();
            // Отцентрировать окно приложения на экране
            setLocation((kit.getScreenSize().width - WIDTH)/2,
(kit.getScreenSize().height - HEIGHT)/2);
            // Установить начальное состояние окна развёрнутым на весь экран
```

```
            setExtendedState(MAXIMIZED_BOTH);

            // Создать меню
            JMenuBar menuBar = new JMenuBar();
            setJMenuBar(menuBar);
            JMenu ballMenu = new JMenu("Мячи");
            Action addBallAction = new AbstractAction("Добавить мяч") {
                  public void actionPerformed(ActionEvent event) {
                        field.addBall();
                        if (!pauseMenuItem.isEnabled() &&
                              !resumeMenuItem.isEnabled()) {
                              // Ни один из пунктов меню не являются
                              // доступными - сделать доступным "Паузу"
                              pauseMenuItem.setEnabled(true);

                        }
                  }
            };
            menuBar.add(ballMenu);
            ballMenu.add(addBallAction);
            JMenu controlMenu = new JMenu("Управление");
            menuBar.add(controlMenu);
            Action pauseAction = new AbstractAction("Приостановить движение"){
                  public void actionPerformed(ActionEvent event) {
                        field.pause();
                        pauseMenuItem.setEnabled(false);
                        resumeMenuItem.setEnabled(true);

                  }
            };
            pauseMenuItem = controlMenu.add(pauseAction);
            pauseMenuItem.setEnabled(false);
            Action resumeAction = new AbstractAction("Возобновить движение") {
                  public void actionPerformed(ActionEvent event) {
                        field.resume();
                        pauseMenuItem.setEnabled(true);
                        resumeMenuItem.setEnabled(false);
                  }
            };
            resumeMenuItem = controlMenu.add(resumeAction);
            resumeMenuItem.setEnabled(false);
            // Добавить в центр граничной компоновки поле Field
            getContentPane().add(field, BorderLayout.CENTER);
      }

      // Главный метод приложения
      public static void main(String[] args) {
            // Создать и сделать видимым главное окно приложения
            MainFrame frame = new MainFrame();
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setVisible(true);
      }
}
```