НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ім. І. СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Кафедра автоматики та управління в технічних системах

Лабораторна робота №1

із дисципліни «Технології паралельних та розподілених обчислень»

на тему «Розробка потоків та дослідження пріоритету запуску потоків»

Підготував:

Студент ФІОТ, гр. ІТ-83,

Троян О.С.

Перевірив:

пос. Дифучина О. Ю.

Київ 2020

Мета роботи: Оволодіти навичками створення та запуск потоків, а також найпростішими засобами управління потоками

Завдання:

1. Реалізуйте програму імітації руху більярдних кульок, в якій рух кожної кульки відтворюється в окремому потоці (див. презентацію «Створення та запуск потоків в java» та приклад). Спостерігайте роботу програми при збільшенні кількості кульок. Поясніть результати спостереження. Опишіть переваги потокової архітектури програм. 10 балів.
2. Модифікуйте програму так, щоб при потраплянні в «лузу» кульки зникали, а відповідний потік завершував свою роботу. Кількість кульок, яка потрапила в «лузу», має динамічно відображатись у текстовому полі інтерфейсу програми. 10 балів.
3. Виконайте дослідження параметру priority потоку. Для цього модифікуйте програму «Більярдна кулька» так, щоб кульки червоного кольору створювались з вищим пріоритетом потоку, в якому вони виконують рух, ніж кульки синього кольору. Спостерігайте рух червоних та синіх кульок при збільшенні загальної кількості кульок. Проведіть такий експеримент: створіть багато кульок синього кольору (з низьким пріоритетом) і одну червоного кольору, які починають рух в одному й тому ж самому місці більярдного стола, в одному й тому ж самому напрямку та з однаковою швидкістю. Спостерігайте рух кульки з більшим пріоритетом. Повторіть експеримент кілька разів, значно збільшуючи кожного разу кількість кульок синього кольору. Зробіть висновки про вплив

пріоритету потоку на його роботу в залежності від загальної кількості потоків. 20 балів.

4. Побудуйте ілюстрацію для методу join() класу Thread з використанням руху більярдних кульок різного кольору. Поясніть результат, який спостерігається. 10 балів.
5. Створіть два потоки, один з яких виводить на консоль символ ‘-‘, а інший – символ ‘|’. Запустіть потоки в основній програмі так, щоб вони виводили свої символи в рядок. Виведіть на консоль 100 таких рядків. Поясніть виведений результат. 10 балів. Використовуючи найпростіші методи управління потоками, добийтесь почергового виведення на консоль символів. 15 балів.
6. Створіть клас Counter з методами increment() та decrement(), які збільшують та зменшують значення лічильника відповідно. Створіть два потоки, один з яких збільшує 100000 разів значення лічильника, а інший – зменшує 100000 разів значення лічильника. Запустіть потоки на одночасне виконання. Спостерігайте останнє значення лічильника. Поясніть результат. 10 балів. Використовуючи синхронізований доступ, добийтесь 8 правильної роботи лічильника при одночасній роботі з ним двох і більше потоків. Опрацюйте використання таких способів синхронізації: синхронізований метод, синхронізований блок, блокування об’єкта. Порівняйте способи синхронізації. 15 балів.

Хід роботи

1. Для кожної кулі виділяється окремий потік який виконує рух кульки до поки вона не опинится у лузі.

```
@Override
public void run() {
    try {
        for (; ;) {
            if (canvas.isInsideAnyHole(b)) {
                canvas.remove(b);
                canvas.repaint();
                return;
            }
            b.move();
            Thread.sleep( millis: 4);
        }
    } catch (InterruptedException ex) {
        System.out.println("Thread " + Thread.currentThread().getName() + " interrupted");
    }
}
```

2. За створення кульок та луз відповідає клас BounceFrame, всі логіка дій прив'язується до клавіш, які разом з активним полем на якому відображається стан кульок, бачить користувач.

2.1. Створення нової кульки:

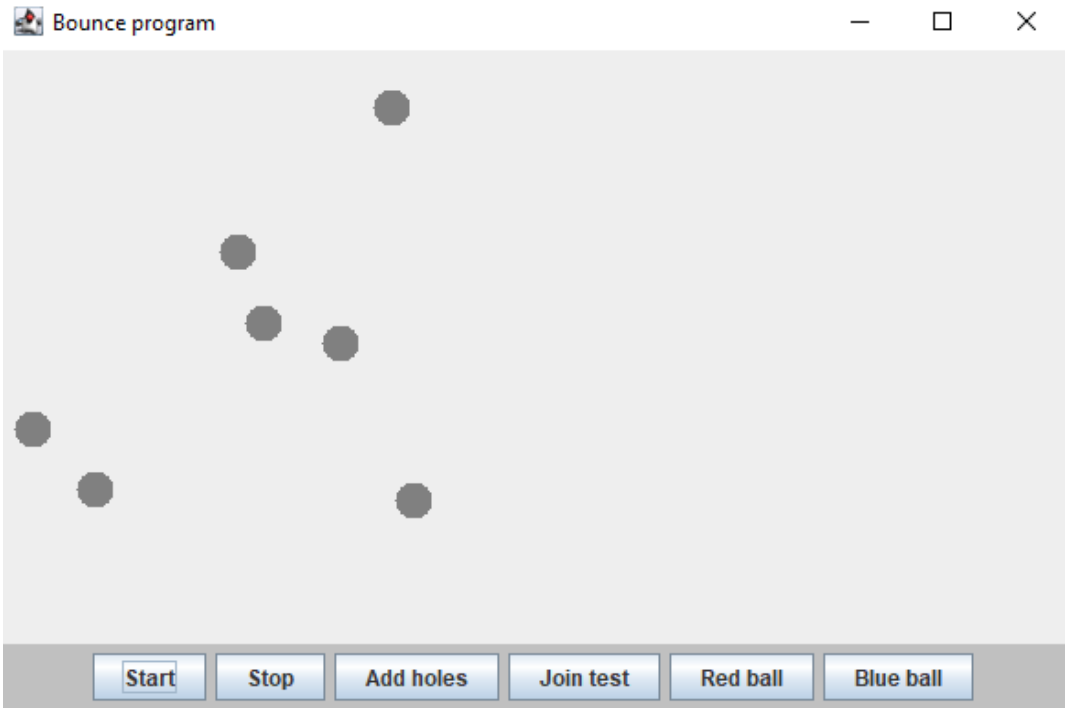
```
var buttonStart = new JButton( text: "Start");
```

```
buttonStart.addActionListener(e -> {
    var b = new Ball(canvas);
    canvas.add(b);
    var thread = new BallThread(b, canvas);
    thread.start();
    System.out.println("Thread name = " + thread.getName());

});
```

```
buttonPanel.add(buttonStart);
```

Створюється нова кулька, відображення якої відбувається в новому потоці.




```
Thread name = main
Thread name = Thread-0
Thread name = Thread-1
Thread name = Thread-2
Thread name = Thread-3
Thread name = Thread-4
Thread name = Thread-5
Thread name = Thread-6
```

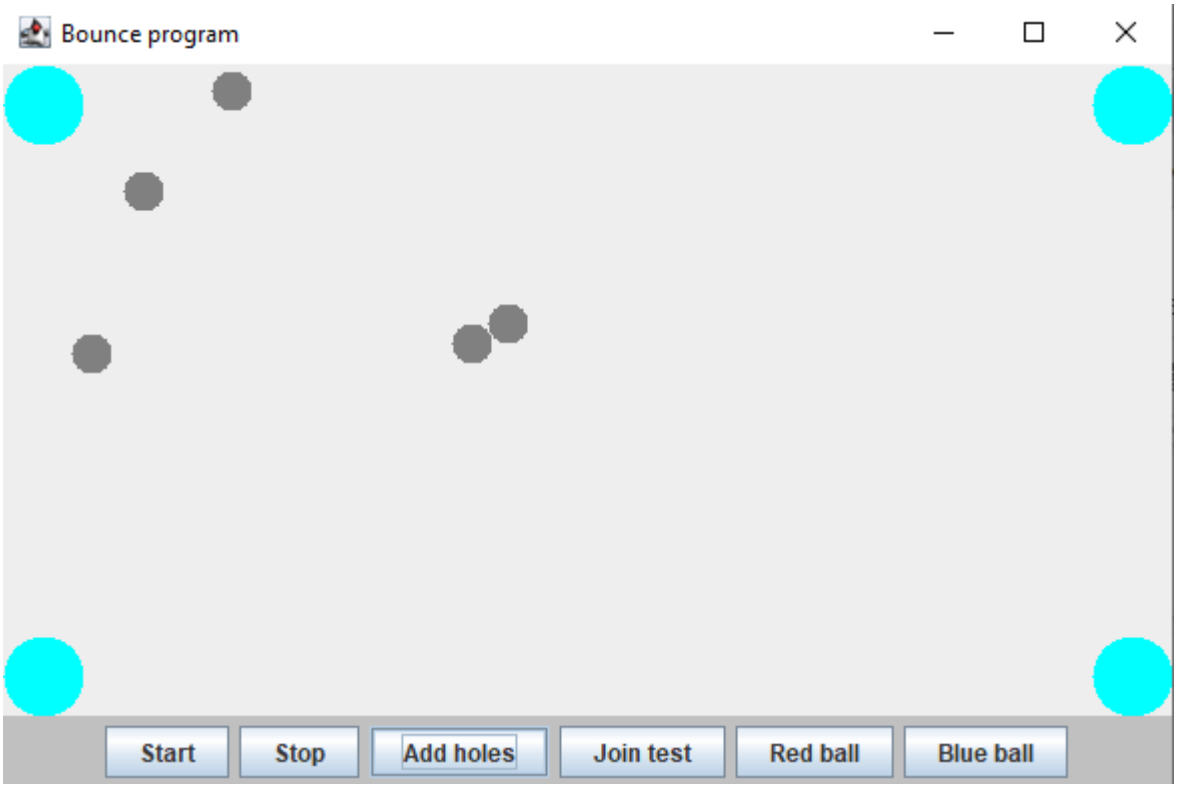
2.2. Створення луз

```
buttonAddHoles.addActionListener(e -> {
    canvas.toggleHoles();
});
```

Лузи створюються по краям активного інтерфейсу:

```
public void toggleHoles() {
    if (this.holes.size() == 0) {
        this.holes.add(new Hole(this, 0, 0));
        this.holes.add(new Hole(this, getWidth() - 40, getHeight() - 40));
        this.holes.add(new Hole(this, getWidth() - 40, 0));
        this.holes.add(new Hole(this, 0, getHeight() - 40));
    } else {
        this.holes.clear();
    }
    this.repaint();
}
```

Якщо вони вже присутні то прибираються.



Кульки же перевіряють чи є точки перетинання з лузами, в позитивному випадку кулька прибирається, і потік завершує свою роботу + збільшує лічильник видалених кульок.

```
public boolean isInsideAnyHole(Ball ball) {
    return holes.stream().anyMatch(hole -> hole.contains(ball.getX(), ball.getY()));
}
```

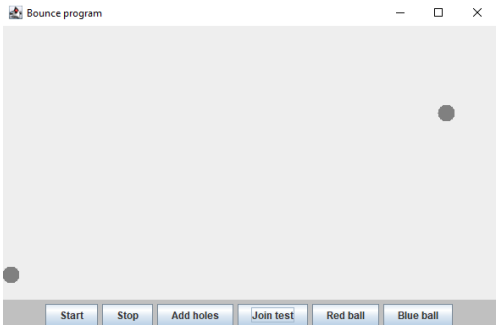
```
1 balls removed
2 balls removed
```

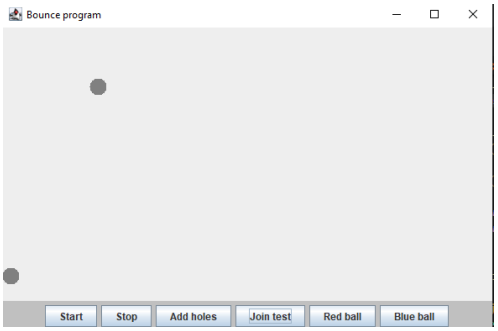
2.3. Join

```
buttonJoin.addActionListener(e -> {
    var b = new Ball(canvas);
    canvas.add(b);
    var c = new Ball(canvas);
    canvas.add(c);
    var thread1 = new BallThread(b, canvas);
    var thread2 = new BallThread(c, canvas);

    new Thread(() -> {
        thread1.start();
        try {
            thread1.join();
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
        thread2.start();
        try {
            thread2.join();
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
    }).start();
});
```

Створюємо 2 кульки то 2 потоки для них, для більш наглядної демонстрації для запуску двох потоків використаємо окремий анонімний потік, це дозволить не реагувати на інші кульки. Запускаємо перший потік, та викликаємо метод .join() , а потім запускаємо інший потік. Поки перший потік не завершить свою роботу, другий не почне свою, тобто поки перша кулька не потрапить в лузу, друга не почне свій рух.



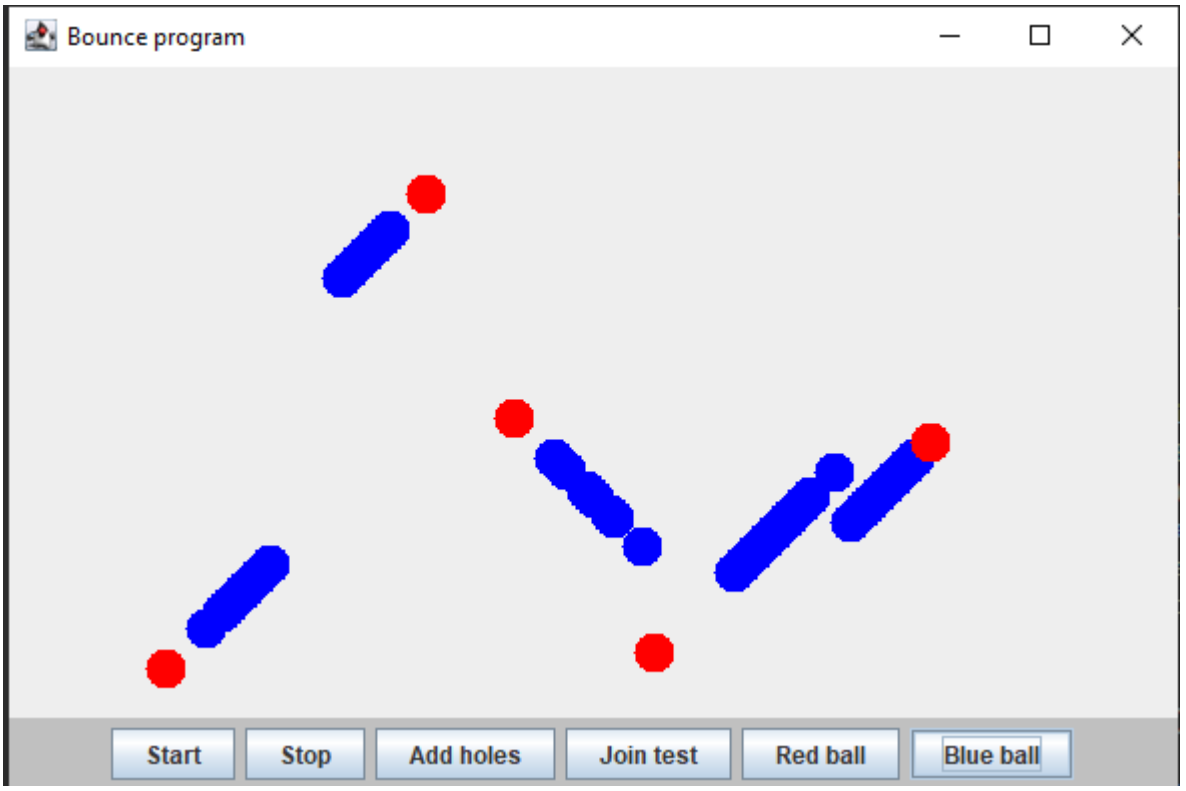


2.4. Створення потоків з різним рівнем пріоріиеиності

```
buttonBlue.addActionListener(e -> {
    int x = 0;
    int y = 30;
    for (int i = 0; i < 10; i++) {
        var b = new Ball(canvas, Color.blue, x, y);
        canvas.add(b);
        var ballThread = new BallThread(b, canvas);
        ballThread.setPriority(Thread.MIN_PRIORITY);
        ballThread.start();
    }

    var r = new Ball(canvas, Color.red, x, y);
    canvas.add(r);
    var redBallThread = new BallThread(r, canvas);
    redBallThread.setPriority(Thread.MAX_PRIORITY);
    redBallThread.start();
});
```

Створюється 10 кульок з малим пріоітетом, та одно за максимальним, при виклиці цього методу декілька разів можливо побачити різницю в пріоритетності.



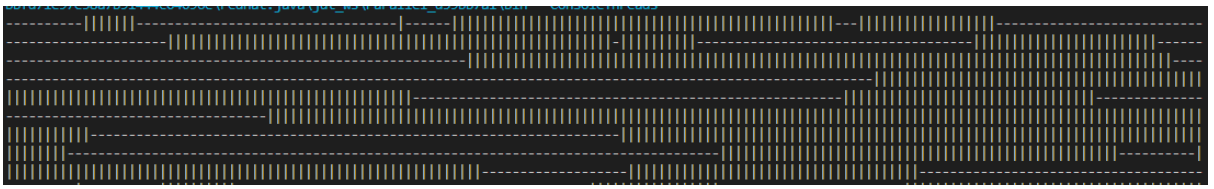
Через вищий пріоритет червона кулька, не дивлячись на те, що створена пізніше за червоні прийде до цілі швидше, так як планувальник потоків використовує пріоритети потоків виконання, щоб прийняти рішення, коли дозволити виконання кожного потоку. Високопріоритетний потік виконання може витісняти фоновий. Наприклад, коли фоновий потік виконується, а високопріоритетний збирається відновити своє виконання, перерване у зв'язку з припиненням або очікуванням завершення операції введення-виведення, то він витісняє фоновий потік.

3. Створюємо два потоки які виводять 2 різні символи:

3.1. Без синхронізацій

```
private static void characters() {
    var firstThread = new Thread(() -> {
        while (true) {
            System.out.print(s: "-");
        }
    });
    var secondThread = new Thread(() -> {
        while (true) {
            System.out.print(s: "|");
        }
    });
    firstThread.start();
    secondThread.start();
}
```

Отримуємо такий результат:



Виводятся символи в довільному порядці тому, що потоки працюють паралельно і незалежно один від одного, тому вгадати в якій послідовності будут виведені символи неможливо.

3.2. Синхронізуємо потоки

```
private static void characters() {
    var sync = new Object();
    AtomicInteger state = new AtomicInteger();
    var firstThread = new Thread(() -> {
        synchronized (sync) {
            while (true) {
                if (state.get() == 0) {
                    System.out.print(s: "-");
                    state.set(newValue: 1);
                    sync.notify();
                } else {
                    try {
                        sync.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }
    });
    var secondThread = new Thread(() -> {
        synchronized (sync) {
            while (true) {
                if (state.get() == 1) {
                    System.out.print(s: "|");
                    state.set(newValue: 0);
                    sync.notify();
                } else {
                    try {
                        sync.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }
    });
    firstThread.start();
    secondThread.start();
}
```

Для синхронізації одночасно було виконано декілька способів. AtomicInteger та synchronized. AtomicInteger - класс який надає доступ до атомарних методів, тому зміна стану буде

виконуватись синхронно. Synchronized за допомогою одного спільного класу object повідомляє кожний з потоків про те що монітор розблокований і може виконатись інший потік.

4. Створимо лічильники для збільшення то зменшення значееня. Для зручності використання патерн стратегія.

```java
public interface Counter {
    public void increment();
    public void decrement();

    public int getValue();
}
```

```java
private static void counter(Counter counter) throws InterruptedException {
    var t1 = new Thread(() -> {
        for (int i = 0; i < 10000; i++) {
            counter.increment();
        }
    });
    var t2 = new Thread(() -> {
        for (int i = 0; i < 10000; i++) {
            counter.decrement();
        }
    });

    t1.start();
    t2.start();
    t1.join();
    t2.join();
    System.out.println("Current thread: " + counter.getValue());
}
```

4.1. Без синхронізації

```java
public class SimpleCounter implements Counter {
    private int counter = 0;

    public void increment() {
        counter++;
    }

    public void decrement() {
        counter--;
    }

    public int getValue() {
        return counter;
    }
}
```

```
Current value: -379
```

Значення не є 0 тому, що потоки одночасно звертаються до одного значення і через це вони не можуть зчитати або ж записати актуальні значення.

4.2. Використання атомарного класу

```java
public class AtomicIntCounter implements Counter {
    private final AtomicInteger integer = new AtomicInteger( initialValue: 0);
    @Override
    public void increment() {
        integer.incrementAndGet();
    }

    @Override
    public void decrement() {
        integer.decrementAndGet();
    }

    @Override
    public int getValue() {
        return integer.get();
    }
}
```

AtomicIntCounter є атомарним классом для роботи з цілочисельним значенням, тому виклик методів буде

відбуватися по черзі і всі операції будуть виконані правильно.

```
Current value: 0
```

4.3. Використання локера

```java
public class LockCounter implements Counter {
    private int counter = 0;
    private final Lock lock = new ReentrantLock();

    @Override
    public void increment() {
        lock.lock();
        counter++;
        lock.unlock();
    }

    @Override
    public void decrement() {
        lock.lock();
        counter--;
        lock.unlock();
    }

    @Override
    public int getValue() {
        return counter;
    }
}
```

Перед виконанням операції локер блокує дії над класом іншим потокам, а потім знову розблоковує, таким чином поки не закінчиться операція в одному потоці другий потік не може почати свою.

```
Current value: 0
```

4.4. Використання synchronized

```java
public class SynchronizedCounter implements Counter {
    private int counter = 0;
    @Override
    public synchronized void increment() {
        counter++;
    }
    @Override
    public synchronized void decrement() {
        counter--;
    }

    @Override
    public int getValue() {
        return counter;
    }
}
```

Synchronized блокує монітор классу тим самим багатопоточний доступ до об'єкта втрачається, після закінчення монітор знову розблоковується і виконується наступна операція.

```
Current value: 0
```

4.5. Використання synchronized через object

```java
public class SynchronizedObjectCounter implements Counter {
    private int counter = 0;

    @Override
    public void increment() {
        synchronized (this) {
            counter++;
        }
    }

    @Override
    public void decrement() {
        synchronized (this) {
            counter--;
        }
    }

    @Override
    public int getValue() {
        return counter;
    }
}
```

Аналогічна логіка до минулого методу, але явно використовується об'єкт який зберігає інформацію монітора.

```
Current value: 0
```

Висновок: Під час виконання лабораторної роботи я навчився створювати потоки та керувати ними. Також було досліджено поведінку потоків залежно від маніпуляцій з ними.