

ЗАНЯТИЕ 1.12

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Тема: Создание многопоточного Java-приложения для сохранения и загрузки объектов

Упражнение №1

В данном упражнении предстоит изучить особенности создания новых потоков в Java. Поток в Java представлен классом **java.lang.Thread**.

Создайте новое java-приложение, добавьте в него класс **MyThread**, который будет расширять стандартный класс **Thread** и переопределите в нем метод **run()**. В теле метода **run** необходимо реализовать код, который будет выводить 10 раз в цикле имя текущего потока.

```
public class MyThread extends Thread{  
  
    @Override  
    public void run() {  
        for(int i = 0; i < 10;i++){  
            System.out.println("Hello! I am " + getName());  
        }  
    }  
}
```

Рисунок 1

В главном классе в методе **main** необходимо создать экземпляр класса **MyThread** и вызвать для него метод **start()**.

```
public static void main(String[] args) {  
    System.out.println("Hello! I am " + Thread.currentThread().getName());  
    Thread th = new MyThread();  
    th.start();  
    System.out.println("Bye! I am " + Thread.currentThread().getName());  
}
```

Этот код
выполняется в
главном потоке

Рисунок 2

Код в методе **main** работает в главном потоке. Код, описанный в методе **run** класса **MyThread**, работает в другом потоке. При этом метод **run** начинает работу после вызова метода **start** (когда именно решает планировщик потоков).

Запустите приложение и обратите внимание на вывод результата в консоль:

```
Hello! I am main
Bye! I am main
Hello! I am Thread-0
Hello! I am Thread-0
Hello! I am Thread-0
Hello! I am Thread-0
Hello! I am Thread-0
Hello! I am Thread-0
Hello! I am Thread-0
Hello! I am Thread-0
Hello! I am Thread-0
Hello! I am Thread-0
Hello! I am Thread-0
```

BUILD SUCCESS

Рисунок 3

Добавьте в методе `main` пометку, что создаваемый поток будет демоном.

```
public static void main(String[] args) {
    System.out.println("Hello! I am " + Thread.currentThread().getName());
    Thread th = new MyThread();
    th.setDaemon(on:true); // указываем что данный поток является демоном
    th.start();
    System.out.println("Bye! I am " + Thread.currentThread().getName());
}
```

Рисунок 4

Сравните результат выполнения приложения с предыдущим.
Попробуйте запустить приложение несколько раз.

```
Hello! I am main
Bye! I am main
Hello! I am Thread-0
Hello! I am Thread-0
Hello! I am Thread-0
Hello! I am Thread-0
Hello! I am Thread-0
Hello! I am Thread-0
Hello! I am Thread-0
```

BUILD SUCCESS

Рисунок 5

После эксперимента строку `th.setDaemon(true)` можно удалить.

Второй способ создания потока – использование интерфейса **Runnable**.

Создайте новый класс – **MyRunnable** и реализуйте в нем интерфейс **Runnable**. Так же необходимо переопределить метод **run**, который будет исполнять код в другом потоке (код можно скопировать из класса **MyThread**, изменив вызов **getName**).

```
public class MyRunnable implements Runnable{

    @Override
    public void run() {
        for(int i = 0; i < 10; i++){
            System.out.println("Hello! I am " + Thread.currentThread().getName());
        }
    }
}
```

Рисунок 6

В главном классе в методе **main** создайте экземпляр класса **MyRunnable** и передайте его в качестве аргумента в конструктор класса **Thread**.

```
public static void main(String[] args) {
    System.out.println("Hello! I am " + Thread.currentThread().getName());
    MyRunnable rn = new MyRunnable();
    Thread th = new Thread(
        System.out.println("Hello! I am " + Thread.currentThread().getName());
    }
}
```

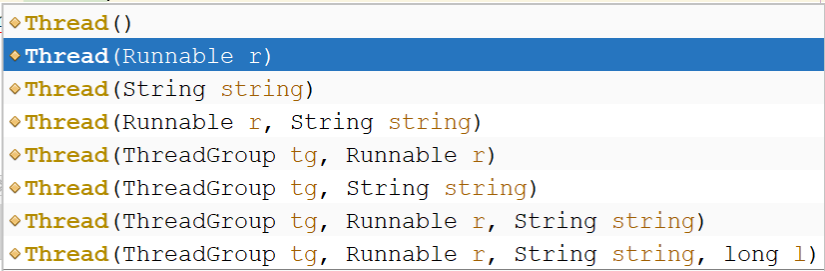


Рисунок 7

Вызовите для потока метод **start**. Убедитесь, что приложение отображает аналогичный первому способу результат.

```
public static void main(String[] args) {
    System.out.println("Hello! I am " + Thread.currentThread().getName());
    MyRunnable rn = new MyRunnable();
    Thread th = new Thread(rn);
    th.start();
    System.out.println("Bye! I am " + Thread.currentThread().getName());
}
```

Рисунок 8

Внесите изменения в метод **run** класса **MyRunnable** так, чтобы имя потока отображалось на консоль с задержкой в 1 секунду. Для этого можно использовать **sleep**.

```

@Override
public void run() {
    for(int i = 0; i < 10; i++){
        System.out.println("Hello! I am " + Thread.currentThread().getName());
        try {
            Thread.sleep(1000); // время в миллисекундах
        } catch (InterruptedException ex) {
        }
    }
}

```

Рисунок 9

Важные моменты в использовании метода `sleep()`:

- Этот метод всегда приостанавливает выполнение текущего потока.
- Фактическое время от остановки потока до пробуждения зависит от системных таймеров и планировщиков. Для не очень загруженной системы, фактическое время для сна будет очень рядом с указанным временем сна. Для загруженной системы этот показатель будет немного больше.
- Любой другой поток может прервать приостановленный поток. В этом случае его выполнение прерывается и выбрасывается исключение (поэтому необходимо использование конструкции `try-catch`).

Запустите программу и убедитесь, что имя потока отображается в цикле 10 раз с интервалом в 1 секунду.

Добавьте в метод `main` вызов метода **`join`**, который позволяет одному потоку ждать завершения другого потока.

```

public static void main(String[] args) {
    System.out.println("Hello! I am " + Thread.currentThread().getName());
    MyRunnable rn = new MyRunnable();
    Thread th = new Thread(rn);
    th.start();
    try {
        th.join(); // теперь главный поток будет ждать окончания работы потока th
    } catch (InterruptedException ex) {
    }
    System.out.println("Bye! I am " + Thread.currentThread().getName());
}

```

Рисунок 10

Теперь главный поток будет ждать окончания работы потока `th`. Запустите программу и сравните полученный результат с предыдущим. Конструкцию с вызовом метода `join` можно удалить.

Одним из способов вызова завершения или прерывания потока представляет метод **interrupt**. Вызов этого метода устанавливает у потока статус, что он прерван. Сам метод возвращает true, если поток может быть прерван, в ином случае возвращается false.

При этом сам вызов этого метода **НЕ завершает поток**, он только устанавливает статус: в частности, метод `isInterrupted` класса `Thread` будет возвращать значение true. Можно проверить значение возвращаемое данным методом и произвести некоторые действия.

Добавьте в метод `main` функционал прерывания потока `th` после 4 секунд работы:

```
public static void main(String[] args) {
    System.out.println("Hello! I am " + Thread.currentThread().getName());
    MyRunnable rn = new MyRunnable();
    Thread th = new Thread(rn);
    th.start();
    try {
        Thread.sleep(4000);
    } catch (InterruptedException ex) {
    }
    th.interrupt();// сообщаем потоку что он должен быть прерван
    System.out.println("Bye! I am " + Thread.currentThread().getName());
}
```

Рисунок 11

Необходимо внести изменения в метод `run` класса `MyRunnable`.

```
public class MyRunnable implements Runnable{

    @Override
    public void run() {
        for(int i = 0; i < 10 && !Thread.currentThread().isInterrupted(); i++){
            System.out.println("Hello! I am " + Thread.currentThread().getName());
            try {
                Thread.sleep(1000); // время в миллисекундах
            } catch (InterruptedException ex) {
                System.out.println(x: "Поток был прерван!");
                Thread.currentThread().interrupt();
            }
        }
    }
}
```

Рисунок 12

Необходимо обратить внимание, что в условии цикла теперь с помощью метода **isInterrupted** проверяется флаг прерван поток или нет. пока этот метод возвращает false, мы можем выполнять цикл. А после того, как будет вызван метод `interrupt`, `isInterrupted` возвратит true, и соответственно произойдет выход из цикла. Однако при получении статуса потока с помощью метода

isInterrupted следует учитывать, что если в цикле обрабатывается исключение **InterruptedException** в блоке catch (в данном случае именно так), то при перехвате исключения статус потока автоматически сбрасывается, и после этого isInterrupted будет возвращать false. Как вариант, в этом случае мы можно повторно прервать текущий поток, вызвав метод interrupt, либо выйти из цикла с помощью break.

Результат работы программы:

```
Hello! I am main
Hello! I am Thread-0
Hello! I am Thread-0
Hello! I am Thread-0
Hello! I am Thread-0
Bye! I am main
Поток был прерван!
-----
BUILD SUCCESS
```

Рисунок 13

Упражнение №2

В данном упражнении необходимо изучить механизм сериализации (десериализации) объектов. Создайте новое приложение. Добавьте в него класс **Person**. Создайте в классе 4 переменные: имя, фамилия, возраст и размер сбережений (бюджет).

Добавьте в класс конструктор, геттеры и сеттеры. Класс должен реализовывать интерфейс **Serializable**.

```
public class Person implements Serializable{

    private String firstName;
    private String lastName;
    private int age;
    private int budget;

    public Person(String firstName, String lastName, int age, int budget) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
        this.budget = budget;
    }

    public int getBudget() {
        return budget;
    }

    public void setBudget(int budget) {
        this.budget = budget;
    }
}
```

Рисунок 14

Создайте класс **SaverRunnable**, реализующий интерфейс **Runnable**. Данный класс будет отвечать за сериализацию объекта типа **Person** в отдельном потоке. Ссылку на **Person** и строковый путь, по которому необходимо сохранить объект оптимально передавать в класс через конструктор класса **SaverRunnable**.

```
public class SaverRunnable implements Runnable{

    private Person person;
    private String path;

    public SaverRunnable(Person person, String path) {
        this.person = person;
        this.path = path;
    }

    @Override
    public void run() {

    }

}
```

Рисунок 15

Переопределите метод **run**. Реализуйте в нем механизм сериализации объекта **Person**. Необходимо использовать **ObjectOutputStream**.

```
@Override
public void run() {
    if(this.path != null && person != null){ // проверка, что параметры не null
        FileOutputStream fos;
        try {
            fos = new FileOutputStream( string:this.path);
            ObjectOutputStream oos = new ObjectOutputStream( out:fos);
            oos.writeObject( obj:person);
            oos.close();
            System.out.println(x:"Success serialization obj Person");
        } catch (IOException ex) {
            System.out.println("Error saving object to path: " + this.path);
        }
    }
}
```

Рисунок 16

В главном классе в методе **main** создайте экземпляр класса **SaverRunnable**, передайте туда ссылку на **Person** и путь к файлу на ПК, в котором необходимо сохранить объект (имя файла может быть любым).

```
public static void main(String[] args) {
    System.out.println("Hello! I am " + Thread.currentThread().getName());
    Person p = new Person( firstName:"Jonh", lastName:"Wick", age:40, budget:100000);
    SaverRunnable sr = new SaverRunnable( person:p, path:"d:\\Work\\person.ser");
    Thread th = new Thread( r:sr);
    th.start();
}
```

Рисунок 17

Запустите программу. Убедитесь, что файл был создан, проверьте его содержимое.

Теперь необходимо реализовать класс, который будет загружать объект из файла. Создайте класс **LoaderRunnable**, так же реализующий Runnable. Для десериализации используется ObjectInputStream.

```
public class LoaderRunnable implements Runnable{

    private String path;

    public LoaderRunnable(String path) {
        this.path = path;
    }

    @Override
    public void run() {
        FileInputStream fis;
        // проверка, что путь не null и файл существует
        if(this.path != null && new File( string:this.path).exists()){
            try {
                fis = new FileInputStream( string:path);
                ObjectInputStream ois = new ObjectInputStream( in:fis);
                Person person = (Person)ois.readObject();
                ois.close();
                System.out.println( x:person.toString());
            } catch (ClassNotFoundException | IOException ex) {
                System.out.println("Error loading object from path: " + this.path);
                System.out.println( x:ex.getMessage());
            }
        }
    }
}
```

Рисунок 18

Для того, чтобы использовать **toString** для объекта Person переопределите его в классе Person:

```
@Override
public String toString() {
    return "Person{first_name = " + this.firstName +
        ", last_name = " + this.lastName +
        ", age = " + this.age +
        ", budget = " + this.budget + "}";
}
```

Рисунок 19

В главном классе в методе main прокомментируйте код, выполнявший сохранение объекта и создайте экземпляр класса LoaderRunnable. В конструктор необходимо передать путь к сохраненному объекту на диске.


```

public static void main(String[] args) {
    System.out.println("Hello! I am " + Thread.currentThread().getName());
    //Person p = new Person("Jonh", "Wick", 40, 100000);
    //SaverRunnable sr = new SaverRunnable(p, "d:\\Work\\person.ser");
    LoaderRunnable lr = new LoaderRunnable(path: "d:\\Work\\person.ser");
    Thread th = new Thread(r:lr);
    th.start();
}

```

Рисунок 20

Запустите программу. В консоли отображается информация об ошибке:
Error loading object from path: d:\Work\person.ser
com.mirea.kt.practical6.Person; local class incompatible: stream classdesc
serialVersionUID = 4257737166262969912, local class serialVersionUID = -
7012698015175706508.

Ошибка (у вас могут быть другие значения) возникла из-за несоответствия версий текущего класса и загружаемого.

Для решения проблемы необходимо привести класс к единой версии и добавить эту информацию в класс Person:

```

public class Person implements Serializable{
    private String firstName;
    private String lastName;
    private int age;
    private int budget;

    private static final long serialVersionUID = 4257737166262969912L;

    public Person(String firstName, String lastName, int age, int budget) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
        this.budget = budget;
    }
}

```

Рисунок 21

Запустите программу и убедитесь, что загрузка (десериализация) объекта произведена успешно.

```

-----
Hello! I am main
Person{first_name = Jonh, last_name = Wick, age = 40, budget = 100000}
-----
BUILD SUCCESS

```

Рисунок 22

Самостоятельно добавьте к переменной бюджет ключевое слово **transient**, заново сохраните и загрузите объект.

```
public class Person implements Serializable{

    private String firstName;
    private String lastName;
    private int age;
    private transient int budget;
}
```

Рисунок 23

Проверьте, как повлияло использование ключевого слова **transient**.

Упражнение №3

Продолжайте работать в этом же приложении. Замените использование интерфейса **Serializable** на **Externalizable**. Также для использования этого интерфейса в сериализуемом классе необходимо создать конструктор без параметров.

Интерфейс **Externalizable** предполагает переопределение методов **writeExternal** и **readExternal**. Это позволяет более гибко реализовать протокол сериализации/десериализации объекта (выбрать нужные поля, использовать дополнительную обработку, например шифрование значений).

```
public Person(String firstName, String lastName, int age, int budget) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.age = age;
    this.budget = budget;
}

public Person() {} // конструктор без аргументов

@Override
public void writeExternal(ObjectOutput out) throws IOException {
    out.writeObject(obj: this.getFirstName());
    out.writeObject(obj: this.getLastName());
    out.writeObject(obj: this.getAge());
    out.writeObject(obj: this.getBudget());
}

@Override
public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
    this.firstName = (String)in.readObject();
    this.lastName = (String)in.readObject();
    this.age = (Integer)in.readObject();
    this.budget = (Integer)in.readObject();
}
```

Рисунок 24

При этом становится важен ПОРЯДОК сериализации/десериализации переменных.

Запустите программу. Убедитесь, что файл успешно загружается.

Индивидуальное задание

Задание: разработать многопоточную консольную Java-программу для сохранения (загрузки) объектов по заданию в соответствии с вариантом.

Требования:

- 1) На сдачу практического задания отводится **7 дней**.
- 2) Вариант определяется согласно порядковому номеру студента в журнале.
- 3) В случае дистанционного выполнения практического задания код программы необходимо выложить на Github и предоставить ссылку на него.
- 4) Итоговая программа должна компилироваться без ошибок, полностью выполнять требуемый функционал и при старте выводить в консоль номер варианта и ФИО студента.
- 5) Названия переменных и методов должны отражать суть и нести смысловую нагрузку.
- 6) Необходимо придерживаться стилистике по написанию Java-кода.
- 7) Необходимо предусмотреть защиту от ввода «аномальных» (ошибочных) значений.
- 8) При выполнении задания необходимо использовать интерфейсы Serializable или Externalizable.

Дополнительная информация:

- 1) Перед выполнением задания рекомендуется ознакомиться с Лекциями №1.9, №1.11.
- 2) Рекомендуется использовать обработку исключений.
- 3) Примерное содержимое консоли после старта программы (в данном случае происходит работа с объектом класса Student):

```
-----[ jar ]-----  
  
--- exec-maven-plugin:3.0.0:exec (default-cli) @ PracticalTask1 ---  
Practical task №11. Variant 1. Student Ivanov A.A. Group ABC 12345  
Enter command number: 1 - create object for class Student, 2 - save object to file, 3 - load object from file, 4 - exit  
1  
Enter student name:  
Alex  
Enter student age:  
22  
Enter student Group:
```

- 4) Если номера версий классов не совпадают, то при десериализации возникнет ошибка java.io.InvalidClassException. Для ее исправления необходимо объявить переменную serialVersionUID в десериализуемом классе и назначить ей значение указанное в ошибке (см. лекцию №1.9).

Вариант 1. Разработать программу для сохранения объекта класса **АВТОМОБИЛЬ (Car)**. Обязательные требования к программе:

- Заполнение полей класса должно осуществляется пользователем после старта программы (используйте класс **Scanner**).
- Класс должен содержать минимум 5 полей (переменных класса).
- Для сериализации необходимо использовать интерфейс **Externalizable**.
- Файл должен быть сохранен в корневой директории локального диска C (название и расширение – любое).
- Процедура сериализации должна быть реализована в отдельном потоке.

В результате работы программы необходимо вывести в консоль путь к сохраненному файлу.

Вариант 2. Разработать программу для открытия и десериализации файла объекта класса **Message** (пакет `com.mirea.kt.example`). Класс имеет следующие поля: `id` (int), `body` (String), `type` (String), `hasAttachments` (boolean), `timestamp` (long). Уникальный идентификатор контроля версий равен: - 3380157693869190848L. Обязательные требования к программе:

- Путь к файлу для десериализации задает пользователь после старта программы.
- Программа должна обрабатывать любой файл, содержащий объект класса **Message** (файл для проверки необходимо получить у преподавателя, <https://goo.su/SRgK>).
- Процедура десериализации должна быть реализована в отдельном потоке.

В результате работы программы необходимо вывести в консоль содержимое полей объекта класса из входного файла.

Вариант 3. Разработать программу для сохранения объекта класса **ВРАЧ (Doctor)**. Обязательные требования к программе:

- Заполнение полей класса должно осуществляется пользователем после старта программы.
- Класс должен содержать минимум 5 полей (переменных класса).
- Файл должен быть сохранен в корневой директории локального диска C (название и расширение – любое).
- Процедура сериализации должна быть реализована в отдельном потоке.

В результате работы программы необходимо вывести в консоль путь к сохраненному файлу.

Вариант 4. Разработать программу для сохранения объекта класса **РАСТЕНИЕ (Plant)**. Обязательные требования к программе:

- Заполнение полей класса должно осуществляться пользователем после старта программы.
- Для сериализации необходимо использовать интерфейс **Externalizable**.
- Класс должен содержать минимум 5 полей (переменных класса).
- Файл должен быть сохранен в корневой директории локального диска C (название и расширение – любое).
- Процедура сериализации должна быть реализована в отдельном потоке.

В результате работы программы необходимо вывести в консоль путь к сохраненному файлу.

Вариант 5. Разработать программу для открытия и десериализации файла объекта класса **Product** (пакет com.mirea.kt.example). Класс имеет следующие поля: code (long), name (String), type (String), isDiscount (boolean), ingredients (ArrayList, содержащий строки), price (double). Уникальный идентификатор контроля версий равен: -3536693998646060163L.

Обязательные требования к программе:

- Путь к файлу для десериализации задает пользователь после старта программы.
- Программа должна обрабатывать любой файл, содержащий объект класса **Product** (файл для проверки необходимо получить у преподавателя, <https://goo.su/P5lqi>).
- Процедура десериализации должна быть реализована в отдельном потоке.

В результате работы программы необходимо вывести в консоль содержимое полей объекта класса из входного файла.