# Лабораторная работа 1. Сборка Java приложений

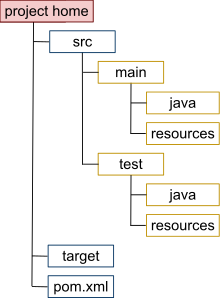
Для разработки на Java требуется пакет инструментов разработчика Java Development Kit (JDK), который состоит из: утилит, стандартных библиотек, документации, примеров, JRE, но не включает в себя среду разработки (Integrated development environment, IDE). Java Runtime Environment (JRE) содержит все, что нужно для запуска скомпилированного приложения: стандартные библиотеки, виртуальную машину (Java Virtual Machine, JVM), утилиту java. Существует несколько версий JDK от различных компаний-вендоров. JDK могут иметь разный набор компонентов, за исключением обязательных. Реализации JVM могут отличаться принципом работы (например, just-in-time или ahead-of-time компиляция), алгоритмом сборщика мусора (Garbage Collector, GC). Также они могут быть платными и бесплатными, с открытым исходным кодом (open source) или проприетарными, с дополнительными условиями использования.

Используемые утилиты:

* javac - компилятор из исходного кода в байт-код. Байт-код - набор инструкций, исполняемых виртуальной машиной. В целом универсален для JVM любой платформы.  
  javac MyClass.java скомпилирует файл MyClass.java в байт-код, сохранит в файл MyClass.class
* jar - инструмент упаковки файлов приложения (байт-кода, ресурсов, дескрипторов) в zip-based архив с расширением .jar.  
  jar cfe app.jar EntryPointClass EntryPointClass.class AnotherClass.class упакует 2 файла байт-кода в jar архив app.jar, сделает его исполняемым, установит класс EntryPointClass, как класс с точкой входа.
* java - загрузчик приложений. Загружает JRE, классы и ресурсы, вызывает точку входа. java EntryPointClass arg1 arg2 запускает байт-код класса EntryPointClass из файла EntryPointClass.class с двумя аргументами java -jar app.jar arg1 arg2 запускает приложение, упакованное в jar архив, с двумя аргументами.
* javap - дизассемблер байт-кода  
  javap -c MyClass.class - дизассемблирует байт-код в файле MyClass.class.
* javadoc - генератор документации на основе комментариев с определенным синтаксисом из исходного кода
* jdb - отладчик приложения (debugger)
* jmc - Java Mission Control - инструмент профилирования и диагностики. Позволяет узнать состояние ресурсов приложения, потоков, сборщика мусора (garbage collector)
* keytool - инструмент работы с ключами шифрования, сертификатами и их контейнерами (keystore)
* и другие ...

### Maven

Apache Maven - фреймворк для автоматизации сборки проектов. В основном Java. Главная цель - облегчить работу с JDK и решить типичные проблемы и задачи сборки. Maven задает файловую структуру проекта, разделяя ресурсы (например, конфиги, шаблоны, скрипты, файлы разметки, шрифты, изображения и прочее), тесты и исходный код.



Настройка сборки проекта через файл pom.xml в корне проекта. Проект можно собрать утилитой mvn. Доступна стационарная и портативная версия (интегрируется в проект) версии утилиты.

#### Жизненный цикл и фазы

Часть процесса сборки можно представить в виде "жизненного цикла" (lifecycle). В maven уже предустановленно 3 lifecycles. Наиболее важный из них - default lifecycle. Lifecycle состоит из последовательности фаз (phase). Наиболее важные фазы default lifecycle в порядке выполнения:

* validate - проверка, что проект корректен и вся необходимая информация доступна
* compile - компилирование исходного кода
* test - запуск юнит-тестов
* package - упаковка скомпилированного кода в архив
* integration-test - запуск интеграционных тестов
* install - регистрация собранного проекта в качестве зависимости в локальном репозитории
* deploy - отправка в удаленный репозиторий и развертывание приложение на сервере

Помимо этого существует clean lifecycle, где наиболее важной фазой является clean - очистка продуктов предыдущего билда.

mvn clean install - означает практически полностью собрать проект. Выполнится clean lifecycle от начала до фазы clean, затем default lifecycle от начала до фазы install (то есть среди перечисленных первой будет фаза validate, последнией будет install, deploy не будет выполнена).

#### POM

Project Object Model (POM) - это xml файл, содержащий информацию о проекте и параметрах сборки. Находится в корне проекта и имеет название pom.xml . Каждый проект maven имеет groupId и artifactId уникальную идентифицирующую пару строк. Обычно в groupId указывают домен в обратном порядке, иногда добавляя группу проектов. Значением artifactId может быть название проекта. Properties - свойства в виде пар ключ-значение. Также указываются зависимости и плагины проекта. Шаблон pom файла:

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>ru.nntu.java-course</groupId>

<artifactId>lab-1</artifactId>

<version>1.0</version>

<properties>

<!-- Измените номер версии используемого JDK, если используется не Java 15 -->

<maven.compiler.target>15</maven.compiler.target>

<maven.compiler.source>15</maven.compiler.source>

</properties>

<dependencies>

<!-- Укажите необходимые зависимости тут -->

</dependencies>

<build>

<!-- Укажите необходимые плагины тут -->

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-assembly-plugin</artifactId>

<executions>

<execution>

<phase>package</phase>

<goals>

<goal>single</goal>

</goals>

<configuration>

<archive>

<manifest>

<mainClass>

<!-- Измените класс с точкой входа в приложения -->

ru.nntu.javacourse1.lab1.App

</mainClass>

</manifest>

</archive>

<descriptorRefs>

<descriptorRef>jar-with-dependencies</descriptorRef>

</descriptorRefs>

</configuration>

</execution>

</executions>

</plugin>

</plugins>

</build>

</project>

#### Зависимости

Зависимость (dependency) - это библиотека, фреймворк или maven проект, который собран с помощью maven и зарегистрирован в хранилище зависимостей. Если создать maven проект, наполнить его контентом, собрать командой mvn clean install, То проект станет зависимостью, будет лежать в локальном репозитории и станет доступнен для включения в другие maven проекты. Локальный репозиторий располагается по пути

Windows: C:\Users\<User\_Name>\.m2

Linux: /home/<User\_Name>/.m2

Mac: /Users/<user\_name>/.m2

Удаленный репозиторий - сервер с коллекцией зависимостей, доступных для загрузки maven. По умолчанию, доступен [Maven Central Repository](https://mvnrepository.com/repos/central). В pom можно указать и другие репозитории. Можно развернуть свой собственный репозиторий. Если в pom указана зависимость, maven в процессе сборки попытается найти ее в локальном репозитории. Если найти не удастся, будет искать в удаленных репозиториях. Если зависимость найдена, добавит копию в локальный. Пример зависимости на библиотеку junit:junit версии 4.12.

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

</dependency>

Maven построен на plugin архитектуре. Любые действия, в том числе и фазы default lifecycle совершаются плагинами. Некоторые плагины включены по умолчанию, даже если их нет в pom.xml (они описаны в родительском super pom). Каждый плагин привязывается к определенной фазе lifecycle. Если необходимо, можно написать собственный плагин, таким образом создавая точку входа в сборочный процесс.

Примеры плагинов:

* maven-compiler-plugin - компилирует исходный код и тесты
* maven-assembly-plugin - собирает исходный код, тесты, ресурсы, документацию в архив. В том числе и jar (по умолчанию не исполняемый).
* surefire - выбирает и запускает юнит-тесты в фазе test.
* failsafe - выбирает и запускает интеграционные тесты в фазе integration-test.
* shade - решает конфликт версий зависимостей с одинаковыми groupId и artifactId, назначая одной из них shade-псевдоним
* aspectj - реализует элементы Аспектно-Ориентированного Программирования (АОП), изменяя исходный код по установленным правилам. Иначе говоря, внедрет "сквозную функциональность". Например, логгирование, бенчмаркинг, транзакции, обработка исключений, кэш, авторизация и проверка прав и прочее.
* jacoco - измеряет степень покрытия кода тестами. Главная метрика - отношение строчек с выражениями, где прошелся хотя бы один тест к общему количеству строчек с выражениями в %

## Предусловия

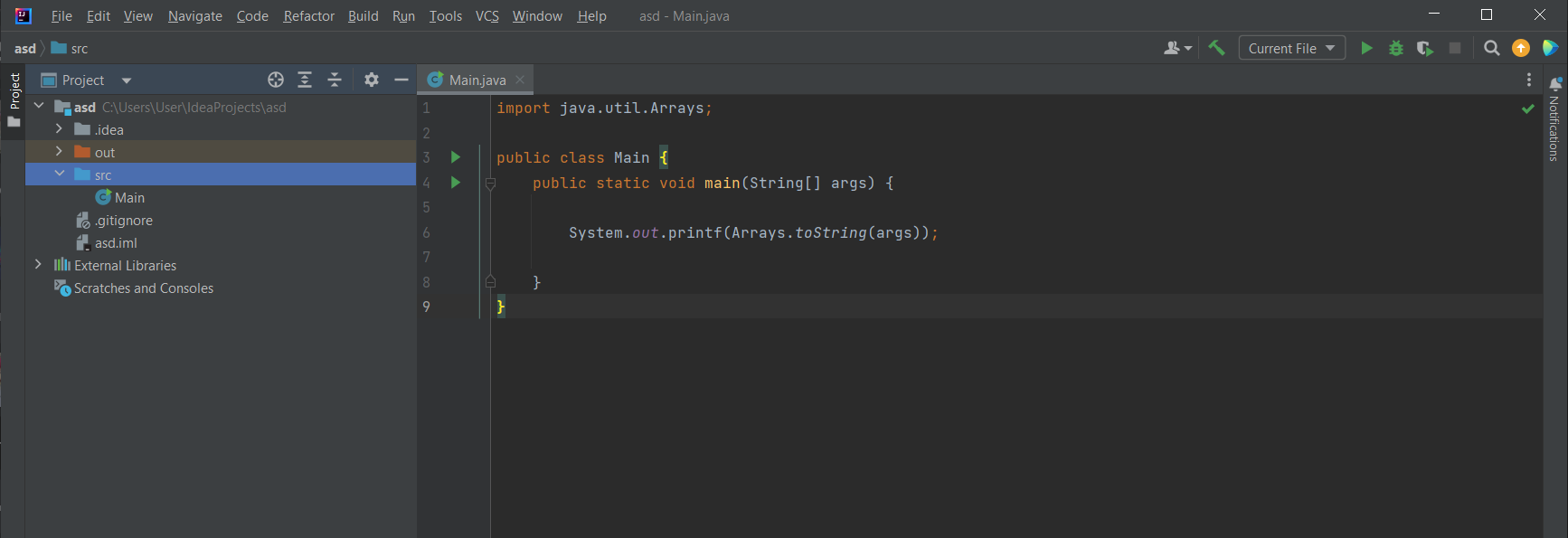
Прежде чем приступать к работе необходимо установить JDK не ниже 8 версии. Подойдет [AdoptOpenJDK](https://adoptopenjdk.net/). Проверить JDK можно командой java --version.

Крайне рекомендуется установить [Intellij IDEA Community](https://www.jetbrains.com/ru-ru/idea/download/) для задач maven и следующих лабораторных работ. Дистрибутив уже содержит maven и мастер создания проекта maven.

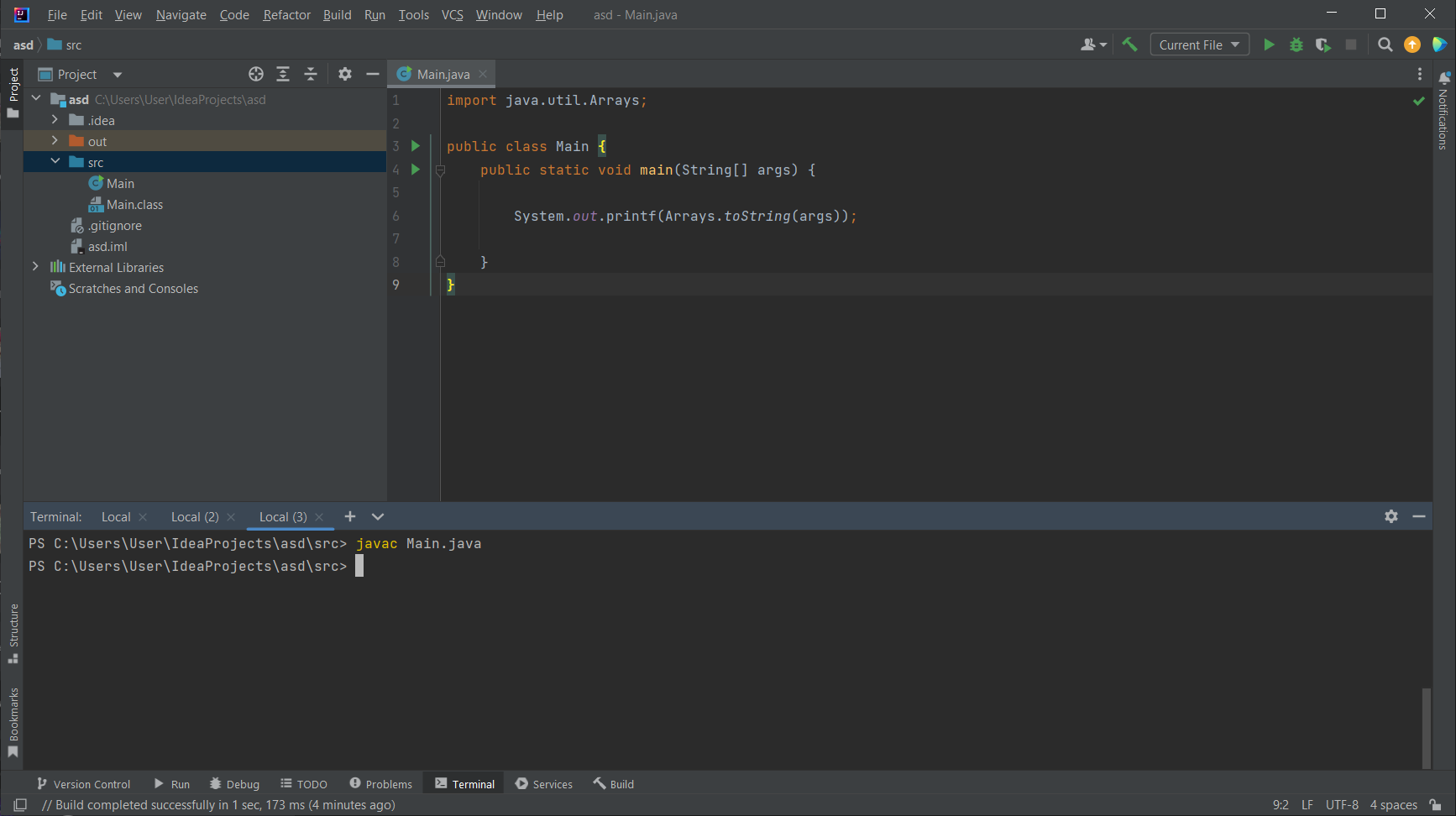
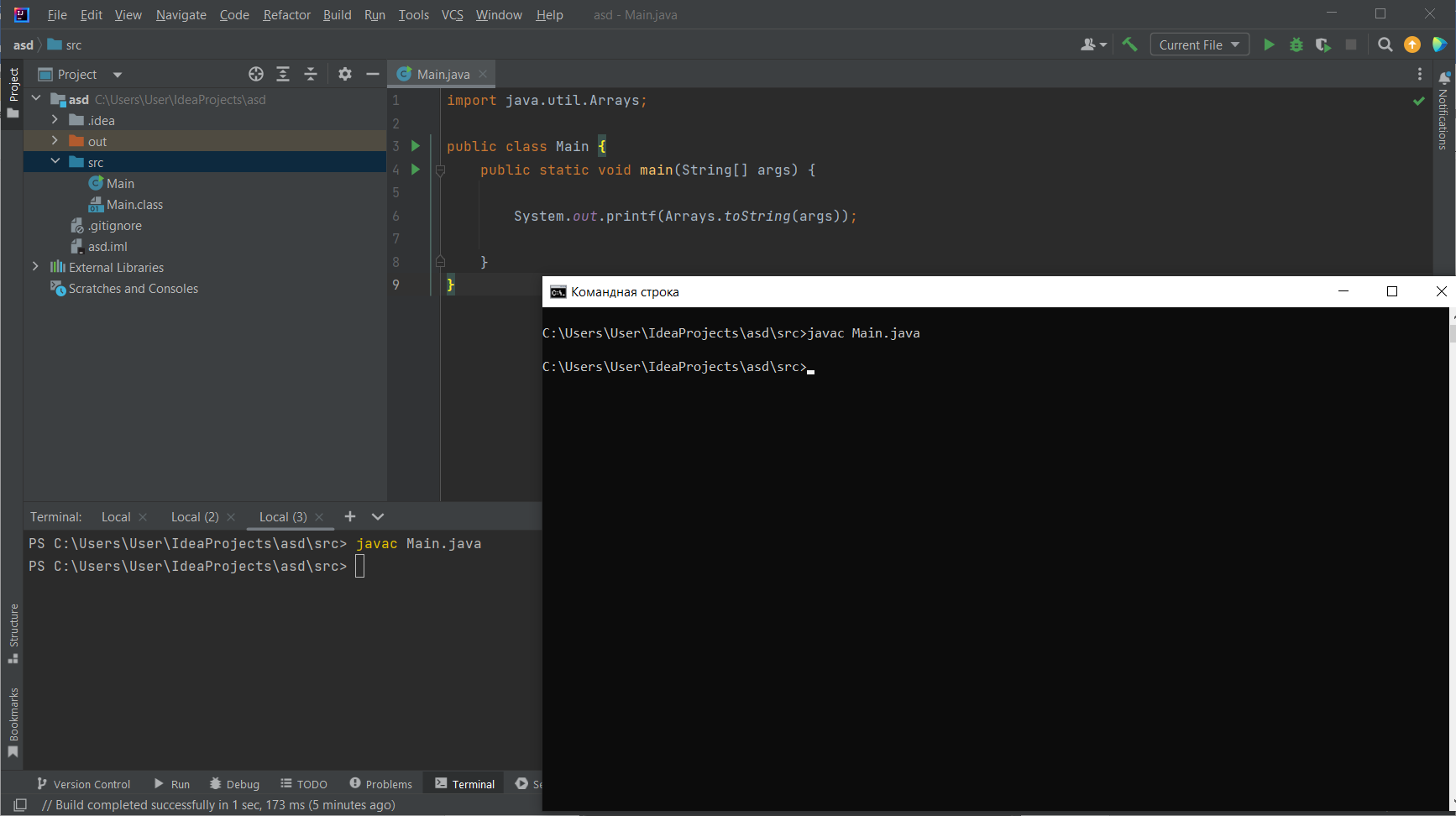
Apache commons-lang3 - набор популярных утилитных методов, в том числе и для работы со строками. [Зависимость в Central Repo](https://mvnrepository.com/artifact/org.apache.commons/commons-lang3), [документация](http://commons.apache.org/proper/commons-lang/apidocs/org/apache/commons/lang3/StringUtils.html), [исходный код](https://github.com/apache/commons-lang).

## Задача

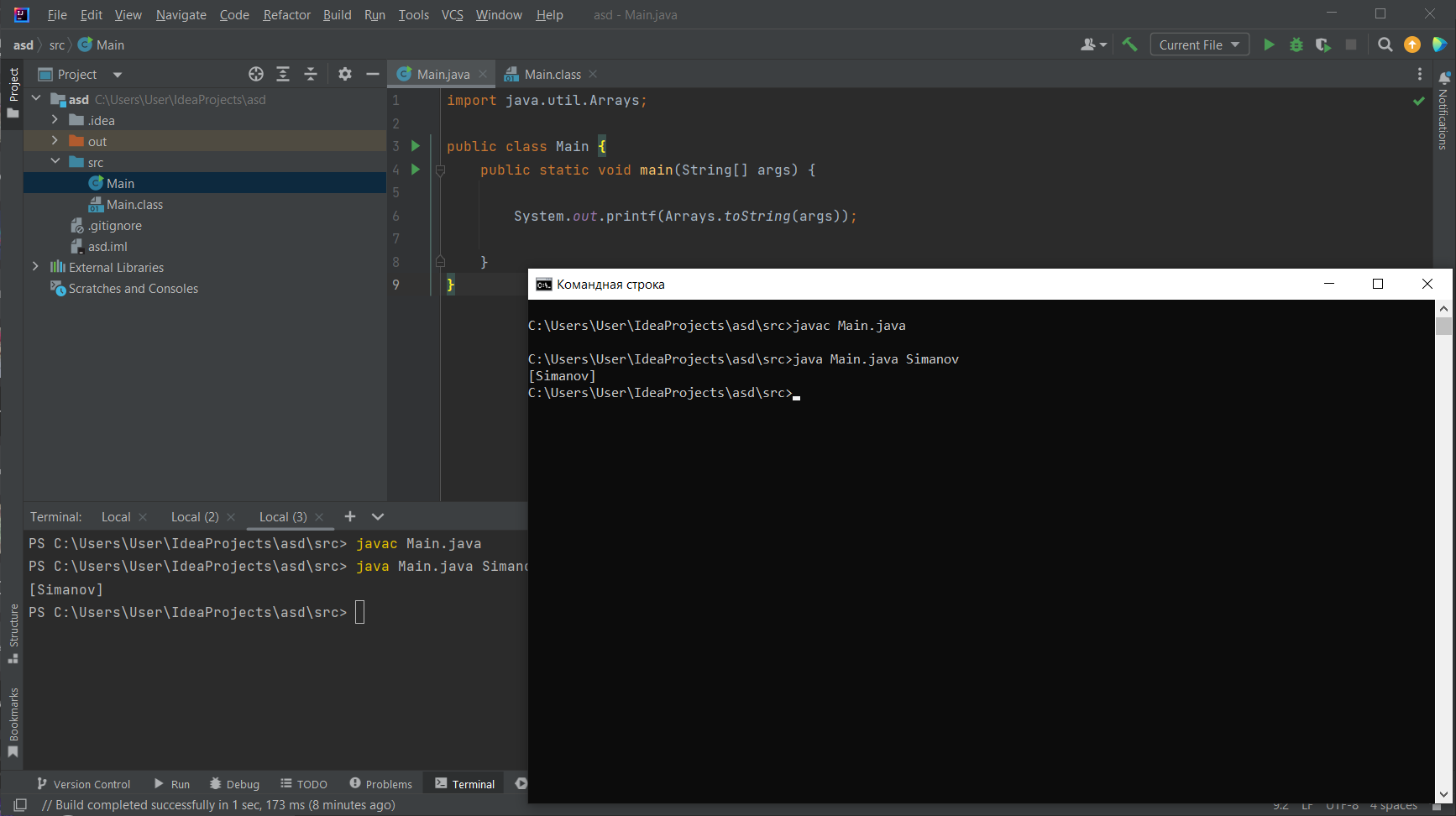
1. Создайте класс с точкой входа в приложение. Главный метод должен выводить в stdout первый аргумент приложения



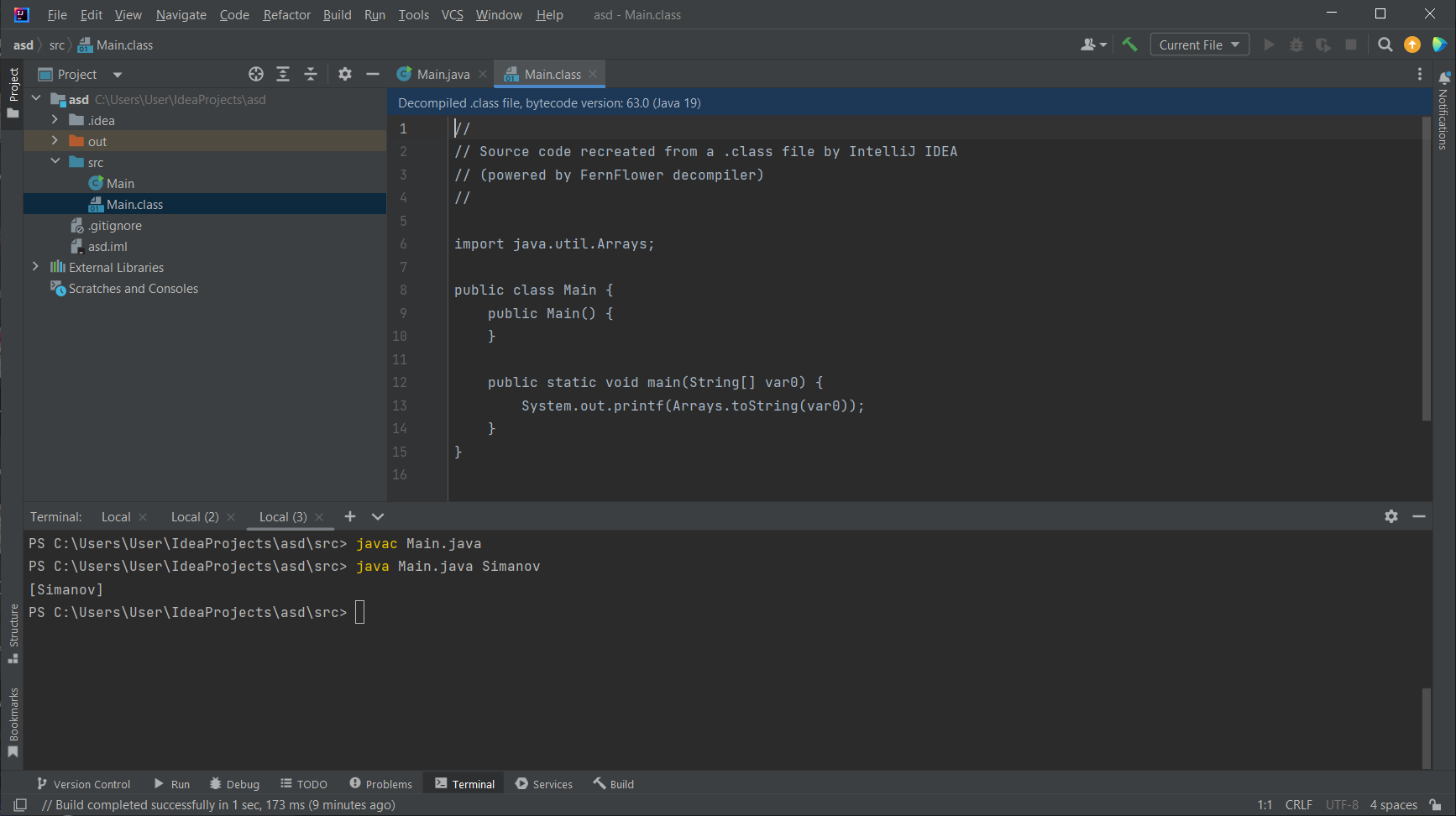
1. Скомпилируйте класс в байт-код



1. Запустите скомпилированный байт-код. В качестве первого аргумента укажите Вашу фамилию



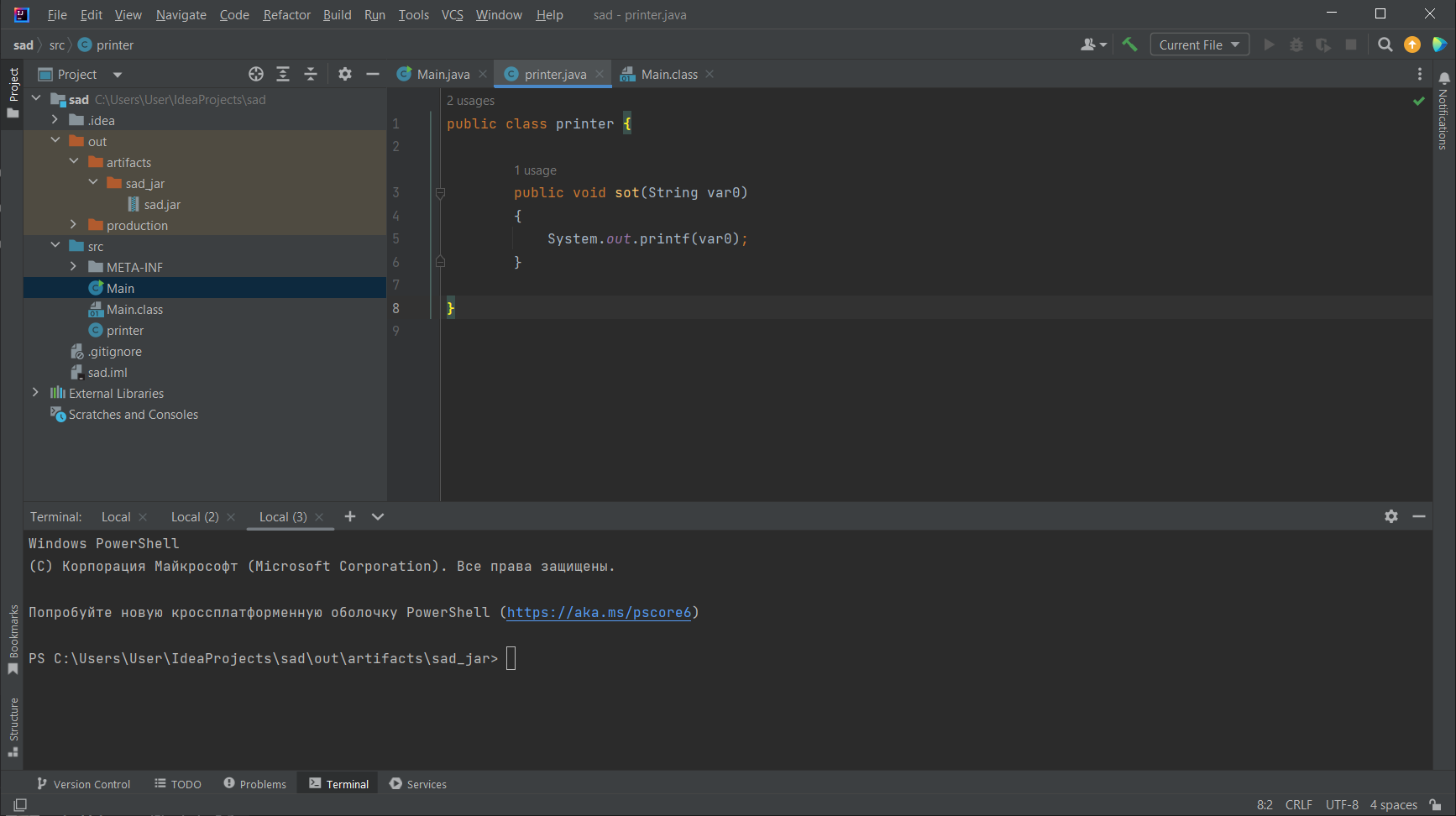
1. Полностью дизасссемблируйте скомпилированный байт-код, включая тело главного метода.



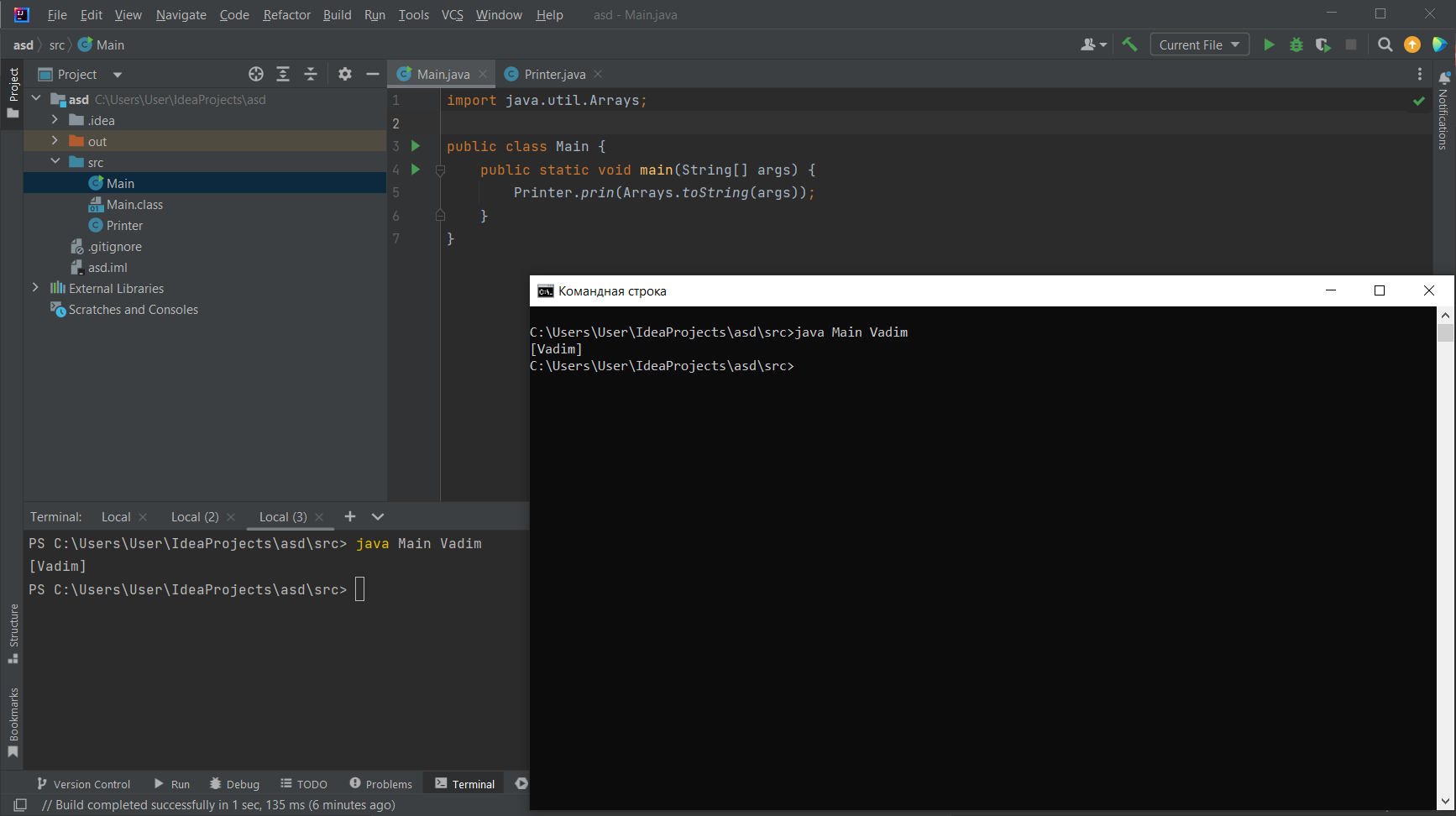
1. Оцените результат, сравните с исходником.

//

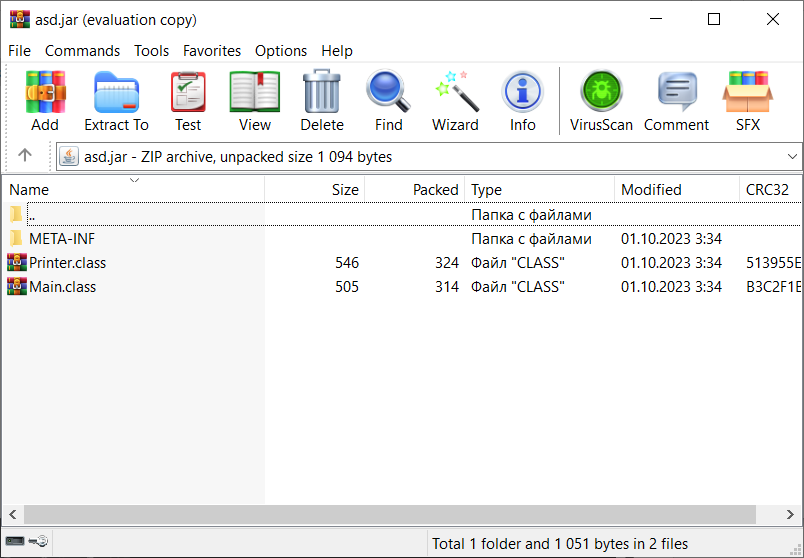
1. Создайте класс, ответственный за печать сообщений. Добавьте метод, который принимает на вход строчку, выводит ее в stdout, и ничего не возвращает



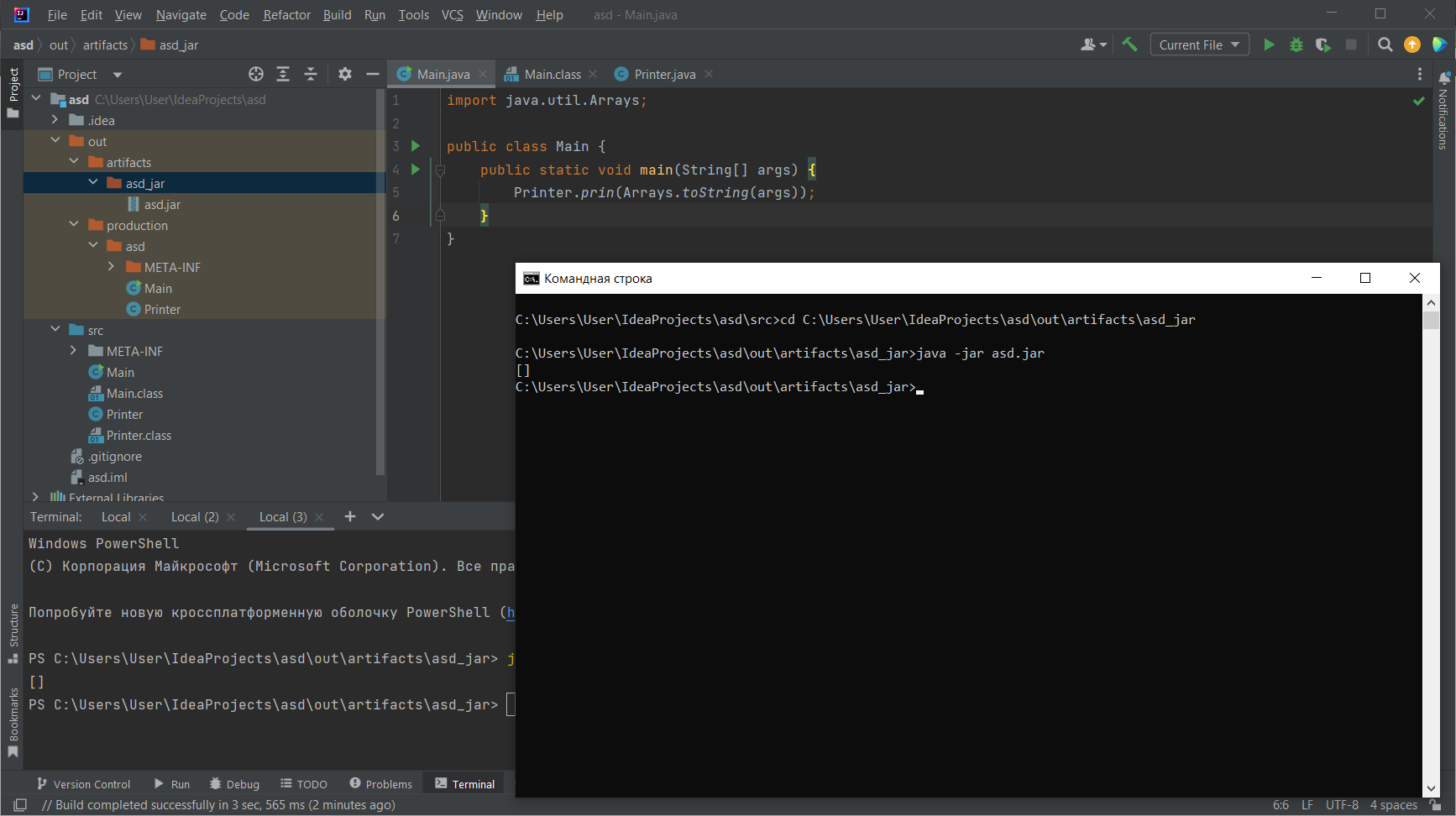
1. Скорректируйте главный класс из шага 1 так, чтобы для печати сообщения он использовал класс-принтер из шага 6.



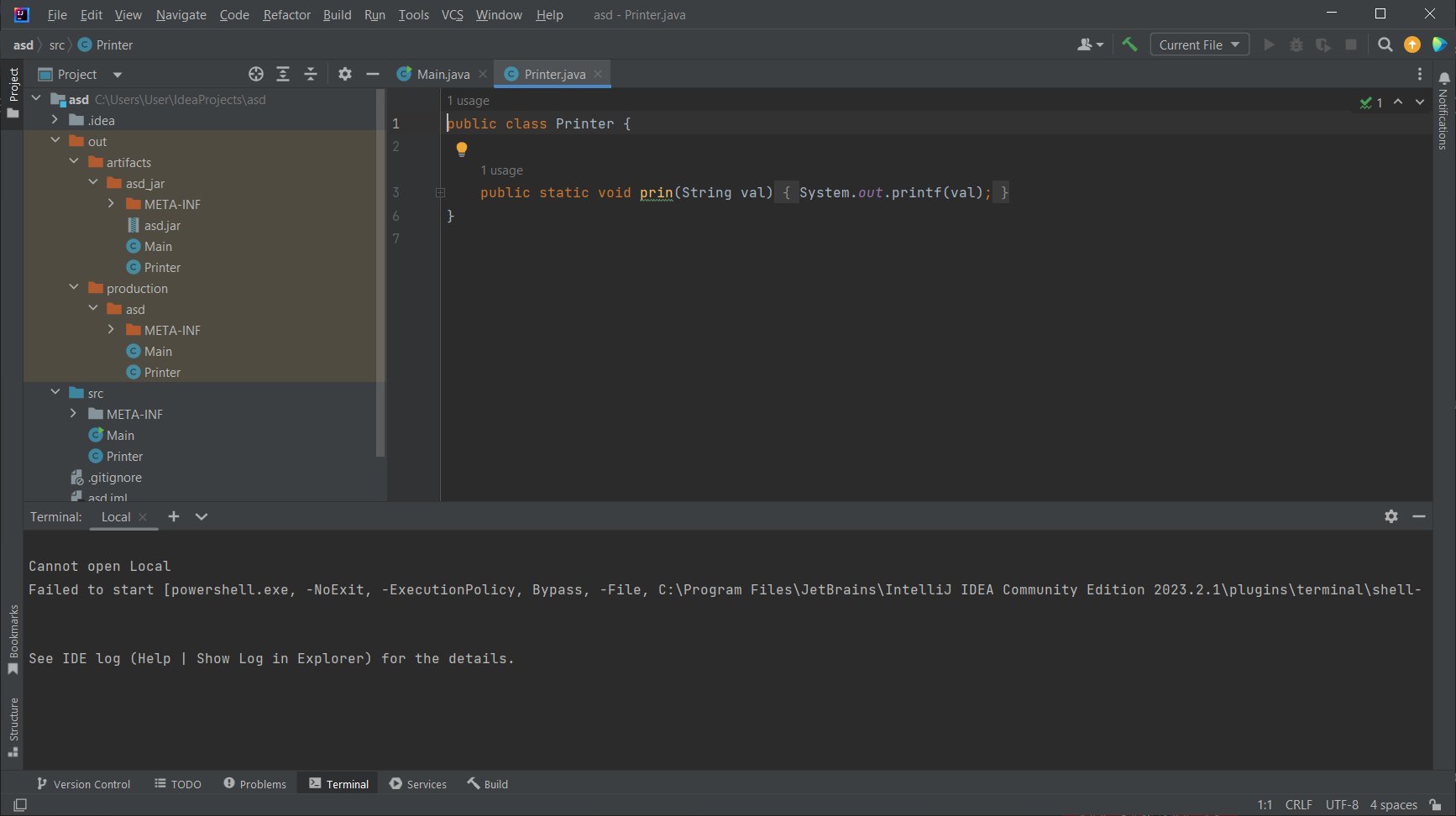
1. Упакуйте оба класса в исполняемый jar.



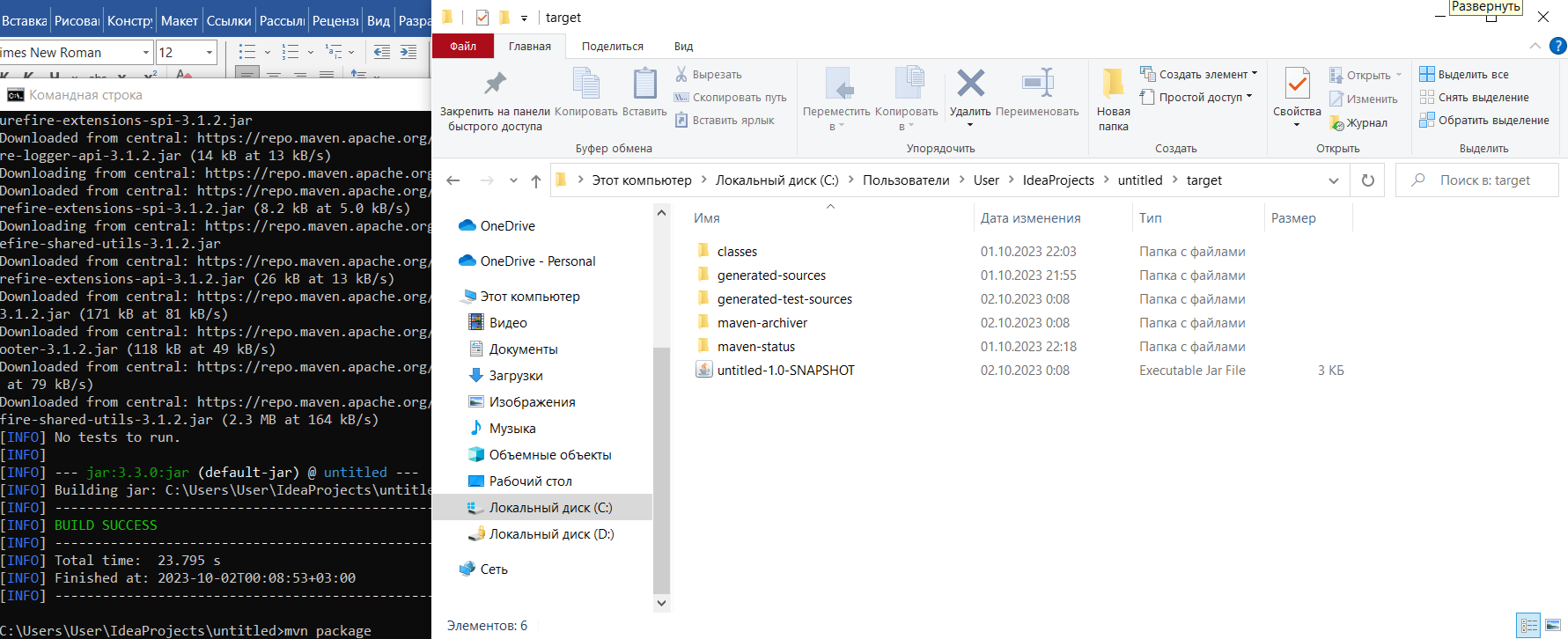
1. Запустите собранный jar. В качестве первого аргумента укажите Ваше имя.



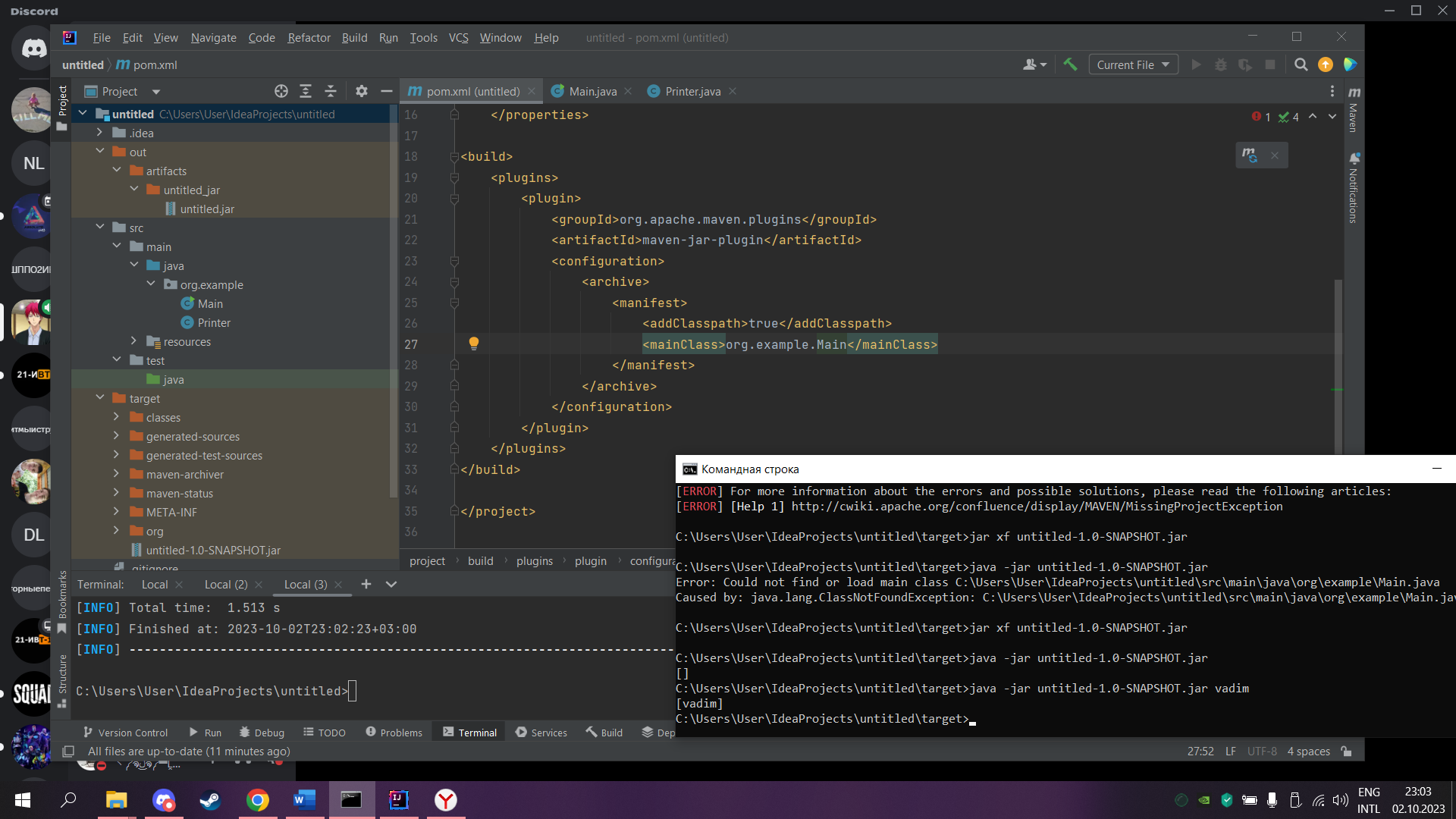
1. На основе этих двух классов создайте maven проект.



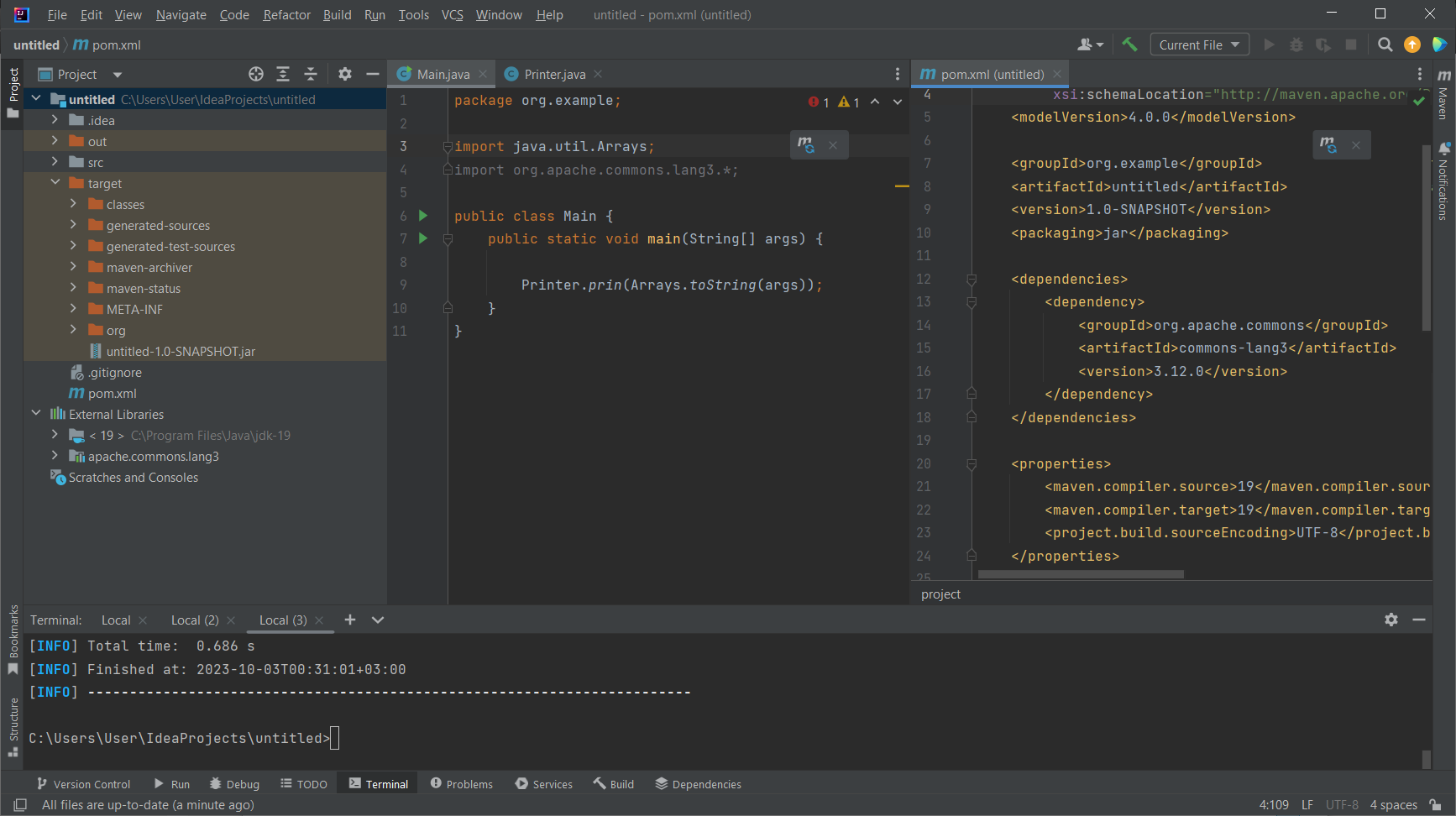
1. Соберите проект в исполняемый jar



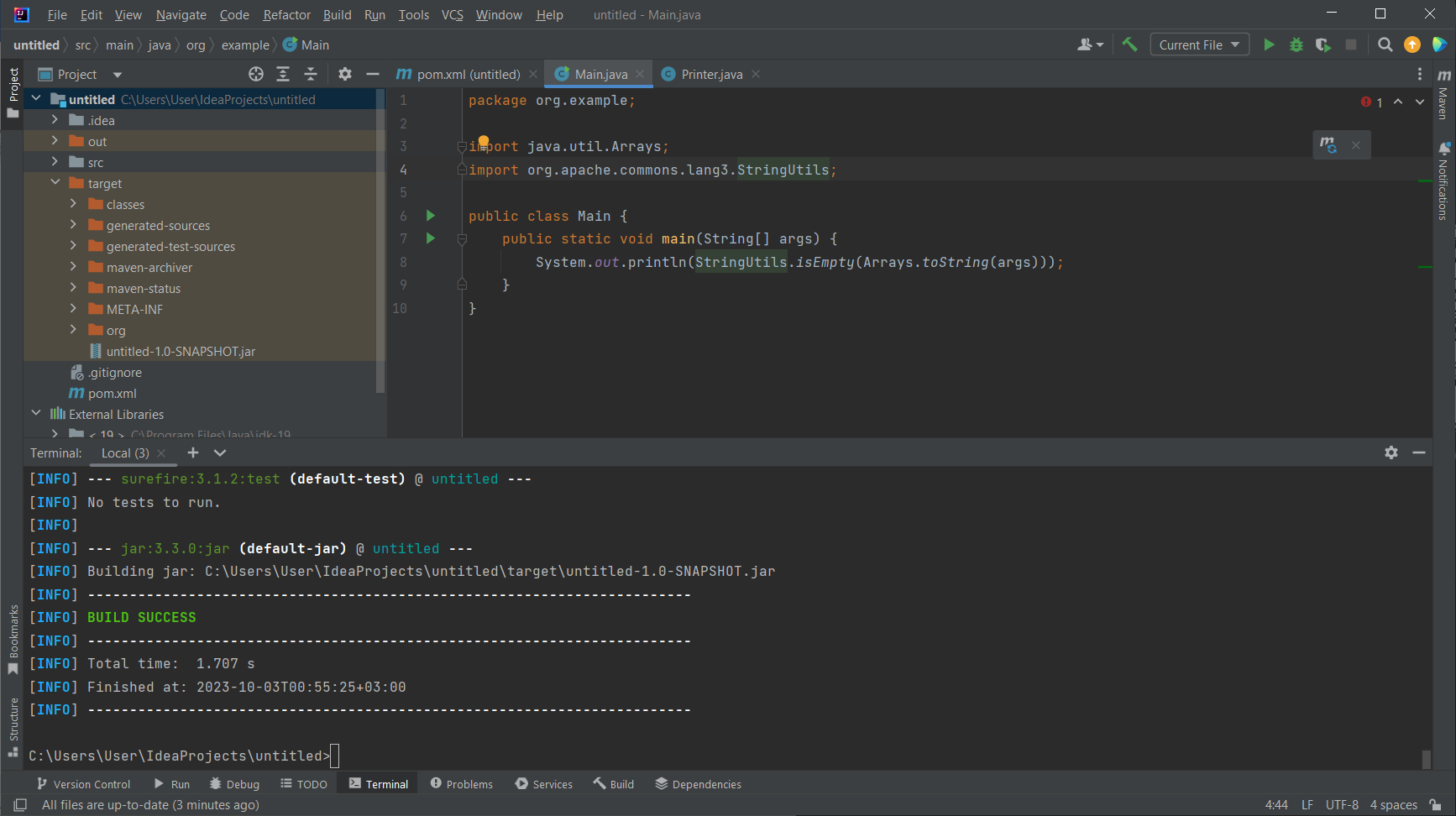
1. Запустите собранный jar.



1. Сравните оба подхода сборки Java приложений.
2. Добавьте зависимость на библиотеку Apache commons-lang3 из Central Repository.



1. Найдите в документации библиотеки класс StringUtils. Выберите любой метод и используйте его в текущем проекте над первым аргументом. Результат работы метода должен отразиться в stdout



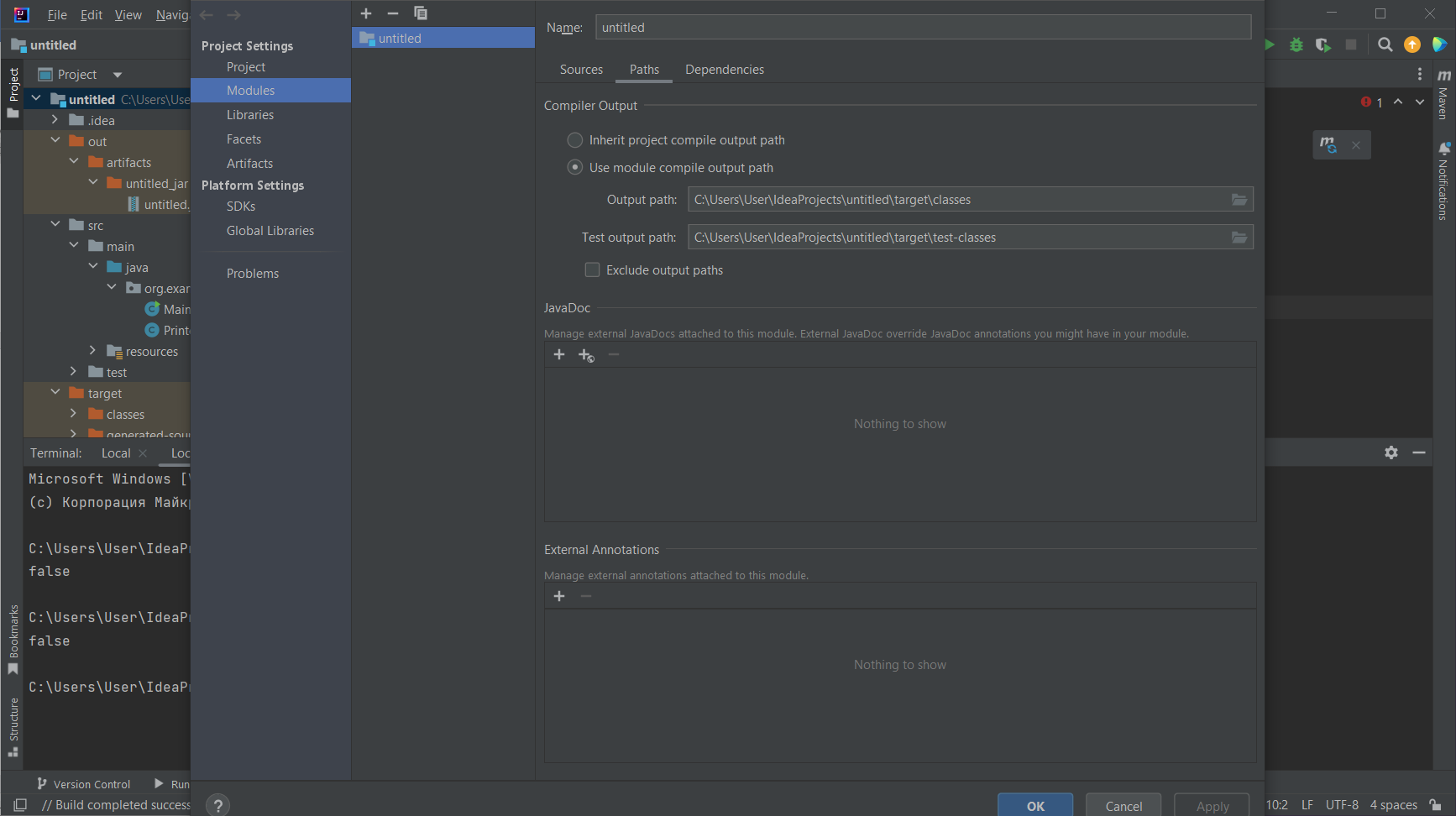
1. Соберите проект в исполняемый jar

Mvn compile

Mvn package

Jar xf имя.jar

1. Запустите собранный jar, проверяя работоспособность выбранного метода из commons-lang3



# ЛАБОРАТОРНАЯ РАБОТА № 2. DEPENDENCY INJECTION.

Необходимо разработать приложение на языке Java с использованием Dependency Injection framework Spring (http://www.springsource.org/spring-framework).

При разработке необходимо использовать Maven.

Для хранения данных используются runtime-коллекции Java в качестве БД (ORM).

Приложение должно представлять из себя обычное консольное приложение, выполняющее одну из следующих функций над некоторой сущностью (см. ниже)

Атрибуты сотрудника, которые должны храниться:

* ФИО
* Адрес проживания
* Принадлежность к отделам
* Дата рождения
* Должность

Атрибуты отдела, которые должны храниться:

* Название
* Количество сотрудников
* Список комнат, в которых размещается

Интерфейс приложения – командная строка.

Предлагается использовать списочную архитектуру для формирования репозитория сотрудников и отделов.

Приложение должно поддерживать:

* Добавление, удаление и поиск записи об отделе
* Добавление, удаление и поиск записи о сотруднике

Приложение должно содержать следующие слои и компоненты

Слои.

1. Слой представления (вывод/ввод результатов на консоль пользователю)
2. Слой Application Services. Здесь должны быть реализованы функции выполнения.
3. Слой домена. Здесь находится определение сущности и репозитория.

Каждый слой должен быть упакован в своем пакете.

При этом должны быть задействованы следующие компоненты:

1. ApplicationService - компонент в котором выполняются функция выполнения. Находится в слое Application Services. В этой компоненте запрещается использовать вывод или ввод в консоль, любые взаимодействия осуществлять через входные/выходные параметры. Все взаимодействия с ORM должны быть осуществлены через соответствующие методы репозитория!

2. Repository -  компонент, инкапсулирующий работу с ORM в виде CRUD  операций (CREATE, REFRESH, UPDATE,DELETE)

Dependency Injection должен склеивать Все компоненты воедино.

При разработке компонент необходимо выставить scope равным Singleton для сервисов и @Repositlry для репозитория.

Для резистрации бинов в Spring context необходимо использовать Java Config classes и auto-scan аннотацию. Проект **может содержать XML**определение бинов контекста.

Также **необходимо**использовать декларативную модель транзакций (через аннотацию @transactional) и**обернуть транзакцией каждый метод из ApplicationService** .

**Пример**

Предположим, заданы:

**сущность** - монитор

**функция выполнения**  - удалить сущность у которой имя начинается на "S"

Приложение должно содержать следующие файлы

* Monitor.java -  содержит класс описания некоторого монитора. Пакет к котором находится сущность - com.university.domain.
* MonitorRepository.java -  интерфейс в котором определены CRUD  методы
* MonitorRepositoryImpl.java -  класс с реализацией репозитория и его CRUD методов
* ApplicationService.java  интерфейс в котором определен метод "удалить сущность у которой имя начинается на "S""
* ApplicationServiceImpl.java  класс с реализацией метода "удалить сущность у которой имя начинается на "S""
* Main.java -  главный файл приложения, в котором происходит инициализация Spring DI. и осуществляется запуск ApplicationService.
* AppConfig.java - Java Config  файл с точкой связывания всех бинов (компонент)

**Monitor.java**

package com.university.domain;

class Monitor{

    private long id; // primary key

    private String name; // model name

    private float size; / screen size in inches ''17, ''10.1 and so on

    // get set section

    public long getId(){return id;}

    public void setId(long id){ this.id = id; }

    public long getName(){return name;}

    public void setName(lString name){ this.name = name; }

    public float getSize(){return size;}

    public void setSize(float size){ this.size = size; }

}

**MonitorRepository.java**

package com.university.domain;

public interface MonitorRepository{

     Monitor getEntity(long id);

     void editMonitor(Monitor updatedEntity);

     void removeMonitor(Monitor updatedEntity);

     List<Monitor> getMonitors();

}

**MonitorRepositoryImpl.java**

package com.university.domain;

@Repository

public class MonitorRepositoryImpl implements MonitorRepository{

... Здесь находится ваша имплементация интерфейса  MonitorRepository

}

**ApplicationService.java**

package com.university.application;

public interface **ApplicationService**{

    void removeEntitiesWithSChar();

}

**ApplicationServiceImpl.java**

package com.university.application;

@Service

public class **ApplicationServiceImpl**{

    @Autowire

    MonitorRepository repository;

    @Override

    public void removeEntitiesWithSChar(char ch){

       List<Monitor> list = repository.getMonitors();

       for(Monitor monitor : list){

           if( monitor.getName().charAt(0)==ch )

               repository.removeMonitor(monitor);

       }

    }

}

**AppConfig.java**

@Configuration

@ComponentScan("com.univercity")

public class AppConfig {

   /\* Example how to register external beans like TransactionManager or SessionFactory etc.

   @Bean

   public HelloWorld helloWorld(){

      return new HelloWorld();

   } \*/

}

**Main.java**

package com.univercity.presentation;

class Main{

   public static main(){

        JavaConfigApplicationContext applicationContext = new JavaConfigApplicationContext(AppConfig.class);

        ApplicationService service = (ApplicationService ) applicationContext.getBean(ApplicationService.class);

        service.removeEntitiesWithSChar('S');

   }

}

Результаты лабораторной работы должны быть оформлены. Должны быть приложены - описание задачи текст программы и тестов.

# Лабораторная работа № 3. Знакомство с Spring JPA

Необходимо разработать приложение на языке Java с использованием на основе приложения, разработанного в рамках Лабораторной работы № 1. Для хранения данных необходимо использовать СУБД PostgreSQL. Доступ к данным осуществляется с помощью Spring JPA. В приложении должны храниться атрибуты сотрудника, атрибуты отдела, данные о начальнике отдела, данные о филиалах.

Атрибуты сотрудника (Staff):

* ФИО
* Адрес проживания
* Принадлежность к отделам
* Дата рождения
* Должность

Атрибуты отдела (Department):

* Название
* Количество сотрудников
* Список комнат, в которых размещается
* Ссылка на начальника отдела (из табл. Staff)

Атрибуты филиала (Organisation):

* Название
* Адрес
* Ссылка на начальника филиала (из табл. Staff)

Интерфейс приложения – командная строка.

Приложение должно поддерживать:

* Добавление, удаление и поиск записи об отделе
* Добавление, удаление и поиск записи о сотруднике
* Добавление, удаление и поиск записей о филиале

Приложение должно содержать следующие слои и компоненты

Слои.

1. Слой представления (вывод/ввод результатов на консоль пользователю)
2. Слой Application Services. Здесь должны быть реализованы функции выполнения.
3. Слой домена. Здесь находится определение сущности и репозитория.

Каждый слой должен быть упакован в своем пакете.

При этом должны быть задействованы следующие компоненты:

1. ApplicationService - компонент в котором выполняются функция выполнения. Находится в слое Application Services. В этой компоненте запрещается использовать вывод или ввод в консоль, любые взаимодействия осуществлять через входные/выходные параметры. Все взаимодействия с ORM должны быть осуществлены через соответствующие методы репозитория!

2. Repository -  компонент, инкапсулирующий работу с ORM в виде CRUD  операций (CREATE, REFRESH, UPDATE,DELETE)

Dependency Injection должен склеивать Все компоненты воедино.

При разработке компонент необходимо выставить scope равным Singleton для сервисов и @Repositlry для репозитория..

# ЛАБОРАТОРНАЯ РАБОТА № 4. ПРОЦЕССЫ, ПОТОКИ И ПРИОРИТЕТЫ

Обычно в многозадачной операционной системе (ОС) выделяют такие объекты, как процессы и потоки.

**Процесс** (process) - это объект, который создается ОС при запуске приложения. Процессу выделяется отдельное адресное пространство, это пространство физически недоступно для других процессов. Процесс может работать с файлами или с каналами связи локальной или глобальной сети.

Для каждого процесса ОС создает один главный **поток** (thread), который является потоком выполняющихся по очереди команд центрального процессора. При необходимости главный поток может создавать другие потоки, пользуясь для этого программным интерфейсом ОС. Все потоки, созданные процессом, выполняются в адресном пространстве этого процесса и имеют доступ к ресурсам процесса.

Если процесс создал несколько потоков, то все они выполняются параллельно, причем время центрального процессора (или нескольких центральных процессоров в многопроцессорных системах) распределяется между этими потоками.

Распределением времени центрального процессора занимается специальный модуль операционной системы - планировщик. Планировщик по очереди передает управление отдельным потокам, так что даже в однопроцессорной системе создается полная иллюзия параллельной работы запущенных потоков.

Распределение времени выполняется для потоков, а не для процессов. Потоки, созданные разными процессами, конкурируют между собой за получение процессорного времени. Каждому потоку задается приоритет его выполнения, уровень которого определяет очередность выполнения того или иного потока.

## Реализация многозадачности в Java

Для создания многозадачных приложений Java необходимо воспользоваться классом **java.lang.Thread**. В этом классе определены все методы, необходимые для создания потоков, управления их состоянием и синхронизации.

Есть две возможности использования класса Thread.

Во-первых, можно создать собственный класс на базе класса Thread и переопределить метод run(). Новая реализация этого метода будет работать в рамках отдельного потока.

Во-вторых, создаваемый класс, может реализовать интерфейс Runnable и реализовать метод run(), который будет работать как отдельный поток.

### *Создание подкласса Thread.* При использовании этого способа для потоков определяется отдельный класс, например:

class myThread extends **Thread** {

**public void run()**  {

// здесь можно добавить код, который будет

// выполняться в рамках отдельного потока

}

// здесь можно добавить специализированный для класса код

}

Метод run**()** должен быть всегда переопределен в классе, наследованном от Thread. Именно он определяет действия, выполняемые в рамках отдельного потока. Если поток используется для выполнения циклической работы, этот метод содержит внутри себя бесконечный цикл.

Метод run() получает управление при запуске потока методом **start()** класса Thread**.** В случае апплетов создание и запуск потоков обычно осуществляется в методе start() апплета.

Остановка работающего потока раньше выполнялась методом **stop()** класса Thread**.** Обычно остановка всех работающих потоков, созданных апплетом, выполняется в методе stop() апплета. Сейчас не рекомендуется использование этого метода. Завершение работы потока желательно проводить так, чтобы происходило естественное завершение метода run. Для этого используется управляющая переменная в потоке.

***Пример. Многопоточное приложение с использованием наследников класса Thread***

// Поток для расчета координат прямоугольника

class ComputeRects extends Thread {

boolean going = true;

// конструктор получает ссылку на создателя объекта - апплет

public ComputeRects(MainApplet parentObj) {

parent = parentObj;

}

public void run() {

while(going) {

int w = parent.size().width-1, h = parent.size().height-1;

parent.RectCoordinates

((int)(Math.random()\*w),(int)(Math.random()\*h));

}

}

MainApplet parent; // ссылка на создателя объекта

}

// Поток для расчета координат овала

class ComputeOvals extends Thread {

boolean going = true;

public ComputeOvals(MainApplet parentObj) {

parent = parentObj;

}

public void run() {

while(going) {

int w = parent.size().width-1, h = parent.size().height-1;

parent.OvalCoordinates

((int)(Math.random()\*w),(int)(Math.random()\*h));

}

}

MainApplet parent; // ссылка на создателя объекта

}

public class MainApplet extends JApplet {

ComputeRects m\_rects = null;

ComputeOvals m\_ovals = null;

int m\_rectX = 0; int m\_rectY = 0;

int m\_ovalX = 0; int m\_ovalY = 0;

// Синхронный метод для установки координат

// прямоугольника из другого потока

public synchronized void RectCoordinates(int x, int y) {

m\_rectX = x; m\_rectY = y;

this.repaint();

}

// Синхронный метод для установки координат овала

// из другого потока

public synchronized void OvalCoordinates(int x, int y) {

m\_ovalX = x; m\_ovalY = y;

this.repaint();

}

@Override

public void start() {

super.start();

// Запускаем потоки

if (m\_rects == null) {

m\_rects = new ComputeRects(this); m\_rects.start();

}

if (m\_ovals == null) {

m\_ovals = new ComputeOvals(this); m\_ovals.start();

}

}

@Override

public void stop() {

super.stop();

// Останавливаем потоки

if(m\_rects != null) m\_rects.going = false;

if(m\_ovals != null) m\_ovals.going = false;

}

public void paint(Graphics g) {

int w = this.getWidth(), h = this.getHeight();

g.clearRect(0, 0, w, h);

g.setColor(Color.red);

g.fillRect(m\_rectX, m\_rectY, 20, 20);

g.setColor(Color.blue);

g.fillOval(m\_ovalX, m\_ovalY, 20, 20);

}

public static void main(String[] args) { }

}

Обратите внимание на запуск и остановку потоков в методах start и stop соответственно, а также на объявление синхронных методов и использованием ключевого слова **synchronized**. Синхронизация крайне важна в многопоточных приложениях, так как потоки, работающие над одними и теми же данными одновременно, могут испортить эти данные.

### *Реализация интерфейса Runnable.* Если нет возможности расширять класс Thread, то можно применить второй способ реализации многозадачности. Допустим, уже существует класс MyClass, функциональные возможности которого удовлетворяют разработчика. Необходимо, чтобы он выполнялся как отдельный поток.

class **MyClass** implements **Runnable** {

// код класса - объявление его элементов и методов

// этот метод получает управление при запуске потока

**public void run()**  {

// здесь можно добавить код, который будет

// выполняться в рамках отдельного потока

}

}

Добавим новый поток в пример 5.1, реализованный через интерфейс Runnable.

***Пример. Доработанное многопоточное приложение***

// Поток для расчета координат линии

class ComputeLines implements Runnable {

boolean going = true;

public ComputeLines(MainApplet parentObj) {

parent = parentObj;

}

public void compute() {

int w = parent.size().width-1, h = parent.size().height-1;

parent.LineCoordinates

((int)(Math.random()\*w),(int)(Math.random()\*h),

(int)(Math.random()\*w), (int)(Math.random()\*h));

}

MainApplet parent; // ссылка на создателя объекта

public void run() {

while(going) { compute(); }

}

}

public class MainApplet extends JApplet {

. . . // скопируйте из предыдущего примера

ComputeLines m\_lines = null;

int m\_lineX1 = 0, m\_lineX2 = 0, m\_lineY1 = 0, m\_lineY2 = 0;

// Синхронный метод для установки координат

// прямоугольника из другого потока

public synchronized void RectCoordinates(int x, int y) {

m\_rectX = x; m\_rectY = y;

this.repaint();

}

// Синхронный метод для установки координат овала

// из другого потока

public synchronized void OvalCoordinates(int x, int y) {

m\_ovalX = x; m\_ovalY = y;

this.repaint();

}

// Синхронный метод для установки координат линии

// из другого потока

public synchronized void LineCoordinates(int x1, int y1, int x2, int y2) {

m\_lineX1 = x1; m\_lineX2 = x2; m\_lineY1 = y1; m\_lineY2 = y2;

this.repaint();

}

@Override

public void start() {

super.start();

// Запускаем потоки

. . . // скопируйте из предыдущего примера

if (m\_lines == null) {

m\_lines = new ComputeLines(this);

new Thread(m\_lines).start();

}

}

@Override

public void stop() {

super.stop();

// Останавливаем потоки

. . . // скопируйте из предыдущего примера

if(m\_lines != null) m\_lines.going = false;

}

public void paint(Graphics g) {

. . . // скопируйте из предыдущего примера

g.setColor(Color.green);

g.drawLine(m\_lineX1, m\_lineX2, m\_lineY1, m\_lineY2);

}

public static void main(String[] args) { }

}

### Применение анимации для мультизадачности

Одним из наиболее распространенных применений апплетов является создание анимационных эффектов типа бегущей строки, мерцающих огней и других эффектов, привлекающих внимание пользователя. Для достижения таких эффектов необходим механизм, позволяющий выполнять перерисовку всего окна апплета или его части периодически с заданным интервалом. Перерисовка окна апплета выполняется методом paint(), который вызывается виртуальной машиной Java асинхронно по отношению к выполнению другого кода апплета, если содержимое окна было перекрыто другими окнами. Для периодической перерисовки окна апплета необходимо создание потока (или нескольких потоков), которые будут выполнять рисование в окне апплета асинхронно по отношению к коду апплета. Например, можно создать поток, который периодически обновляет окно апплета, вызывая для этого метод repaint(), или рисовать из потока непосредственно в окне апплета. Также можно использовать класс таймера (этот механизм уже использовался в предыдущих работах). Таймер, по сути, тоже является потоком, выполняющимся параллельно с основным.

Рассмотрим пример апплета 5.3, который умеет сам себя перерисовывать при помощи дополнительного потока.

***Пример. Самоперерисовывающийся апплет***

public class MainApplet extends JApplet implements Runnable {

boolean m\_isGoing = false;

@Override

public void run() {

while (m\_isGoing) {

repaint();

try { Thread.sleep(500); }

catch(InterruptedException e) { stop(); }

}

}

@Override

public void start() {

super.start();

// Запускаем поток

m\_isGoing = true;

new Thread(this).start();

}

@Override

public void stop() {

super.stop();

// Останавливаем потоки

m\_isGoing = false;

// Дадим потоку время завершиться

try { Thread.sleep(500); }

catch(InterruptedException e) {}

}

public void paint(Graphics g) {

int w = this.getWidth(), h = this.getHeight();

g.setColor(new Color((int)(Math.random() \* 255),

(int)(Math.random() \* 255), (int)(Math.random() \* 255)));

g.fillRect(0, 0, w, h);

}

public static void main(String[] args) { }

}

Класс Thread содержит несколько конструкторов и большое количество методов для управления потоков.

***Некоторые методы класса Thread:***

**currentThread()** - возвращает ссылку на выполняемый в настоящий момент объект класса Thread;

**sleep()** - переводит выполняемый в данное время поток в режим ожидания в течение указанного промежутка времени ;

**start()** - начинает выполнение потока. Это метод приводит к вызову соответствующего метода run();

**run()** - фактическое тело потока. Этот метод вызывает после запуска потока;

**stop()** - останавливает поток (устаревший метод);

**isAlive()** - определяет, является ли поток активным (запущенным и не остановленным);

**suspend()** - приостанавливает выполнение потока (устаревший метод);

**resume()** - возобновляет выполнение потока (устаревший метод). Этот метод работает только после вызова метода suspend();

**setPriority()** - устанавливает приоритет потока (принимает значение от MIN\_PRIORITY до MAX\_PRIORITY);

**getPriority()** - возвращает приоритет потока;

**wait()** - переводит поток в состояние ожидания выполнения условия, определяемого переменной условия;

**join()** - ожидает, пока данный поток не завершит своего существования бесконечно долго или в течении некоторого времени;

**setDaemon()** - отмечает данный поток как поток-демон или пользовательский поток. Когда в системе останутся только потоки-демоны, программа на языке Java завершит свою работу;

**isDaemon()** - возвращает признак потока-демона.

### *Состояние потока*

Во время своего существования поток может переходить во многие состояния, находясь в одном из нижеперечисленных состояний:

* Новый поток
* Выполняемый поток
* Невыполняемый поток
* Завершенный поток

***Новый поток.*** При создании экземпляра потока этот поток приобретает состояние “Новый поток”:

Thread myThread=new Thread();

В этот момент для данного потока распределяются системные ресурсы; это всего лишь пустой объект. В результате все, что с ним можно делать - это запустить: myThread.start**()**;

Любой другой метод потока в таком состоянии вызвать нельзя, это приведет к возникновению исключительной ситуации.

***Выполняемый поток.*** Когда поток получает метод start**()**, он переходит в состояние “Выполняемый поток”. Процессор разделяет время между всеми выполняемыми потоками согласно их приоритетам.

***Невыполняемый поток.*** Если поток не находится в состоянии “Выполняемый поток”, то он может оказаться в состоянии “Невыполняемый поток”. Это состояние наступает тогда, когда выполняется одно из четырех условий:

* *Поток был приостановлен*. Это условие является результатом вызова метода suspend(). После вызова этого метода поток не находится в состоянии готовности к выполнению; его сначала нужно “разбудить” с помощью метода resume(). Это полезно в том случае, когда необходимо приостановить выполнение потока, не удаляя его. Поскольку метод suspend не рекомендуется к использованию, приостановка потока должна выполняться через управляющую переменную.
* *Поток ожидает***.** Это условие является результатом вызова метода sleep(). После вызова этого метода поток переходит в состояние ожидания в течении некоторого определенного промежутка времени и не может выполняться до истечения этого промежутка. Даже если ожидающий поток имеет доступ к процессору, он его не получит. Когда указанный промежуток времени пройдет, поток переходит в состояние “Выполняемый поток”. Метод resume() не может повлиять на процесс ожидания потока, этот метод применяется только для приостановленных потоков.
* *Поток ожидает извещения*. Это условие является результатом вызова методаwait(). С помощью этого метода потоку можно указать перейти в состояние ожидания выполнения условия, определяемого переменной условия, вынуждая его тем самым приостановить свое выполнение до тех пор, пока данное условие удовлетворяется. Какой бы объект не управлял ожидаемым условием, изменение состояния ожидающих потоков должно осуществляться посредством одного из двух методов этого потока - notify() или notifyAll(). Если поток ожидает наступление какого-либо события, он может продолжить свое выполнение только в случае вызова для него этих методов.
* *Поток заблокирован другим потоком*. Это условие является результатом блокировки операцией ввода-вывода или другим потоком. В этом случае у потока нет другого выбора, как ожидать до тех пор, пока не завершится команда ввода-вывода или действия другого потока. В этом случае поток считается невыполняемым, даже если он полностью готов к выполнению.

***Завершенный поток.*** Когда метод run() завершается, поток переходит в состояние “Завершенный поток”.

### *Приоритеты потоков.* В языке Java каждый поток обладает приоритетом, который оказывает влияние на порядок его выполнения. Потоки с высоким приоритетом выполняются чаще потоков с низким приоритетом. Поток наследует свой приоритет от потока, его создавшего. Если потоку не присвоен новый приоритет, он будет сохранять данный приоритет до своего завершения. Приоритет потока можно установить с помощью метода setPriority(), присваивая ему значение от MIN\_PRIORITY до MAX\_PRIORITY (константы класса Thread). По умолчанию потоку присваивается приоритет Thread.NORM\_PRIORITY.

Потоки в языке Java планируются с использованием алгоритма планирования с фиксированными приоритетами.Этот алгоритм, по существу, управляет потоками на основе их взаимных приоритетов, кратко его можно изложить в виде следующего правила: в любой момент времени будет выполняться “Выполняемый поток” с наивысшим приоритетом. Как выполняются потоки одного и того же приоритета, в спецификации Java не описано.

### *Группы потоков.* Все потоки в языке Java должны входить в состав группы потоков. В классе Thread имеется три конструктора, которые дают возможность указывать, в состав какой группы должен входить данный создаваемый поток.

Группы потоков особенно полезны, поскольку внутри их можно запустить или приостановить все потоки, а это значит, что при этом не потребуется иметь дело с каждым потоком отдельно. Группы потоков предоставляют общий способ одновременной работы с рядом потоков, что позволяет значительно сэкономить время и усилия, затрачиваемые на работу с каждым потоком в отдельности.

В приведенном ниже фрагменте программы создается группа потоков под названием genericGroup (родовая группа). Когда группа создана, создаются несколько потоков, входящих в ее состав:

**ThreadGroup** genericGroup=new **ThreadGroup**("My generic group");

Thread t1=new Thread(genericGroup,this);

Thread t2=new Thread(genericGroup,this);

Если при создании нового потока не указать, к какой конкретной группе он принадлежит, этот поток войдет в состав группы потоков main (главная группа). Иногда ее еще называют текущей группой потоков. В случае апплета main может и не быть главной группой. Право присвоения имени принадлежит Web-браузеру. Для того чтобы определить имя группы потоков, можно воспользоваться методом **getName**() класса ThreadGroup.

Для того, чтобы определить к какой группе принадлежит данный поток, используется метод **getThreadGroup()**, определенный в классе Thread. Этот метод возвращает имя группы потоков, в которую можно послать множество методов, которые будут применяться к каждому члену этой группы.

**Программирование движения объекта**

Для того чтобы запрограммировать движение объекта из одной точки в другую, вспомним равномерное движение.

Пусть (x1;y1) – начальная координата объекта, а (x2;y2) – конечная координата.

Направление движения задается вектором (dx;dy) = Нормализация(x2-x1; y2-y1), где

Нормализация(a,b) = (a / sqrt(a^2 + b^2); b / sqrt(a^2 + b^2)).

Таким образом, координаты объекта в момент времени T от начала движения, движущегося со скоростью 10 шагов в секунду по прямой, равны: (xc;yc) = (x1; y1) + (dx;dy) \* 10 шагов в секунду \* T;

Скорость 10 шагов в секунду измеряется в пиксель/сек, т.е. количество пикселей пройденных объектом на 1 сек.

Когда (xc;yc) станет равно (x2;y2), тогда объект дошел до конечной точки. На практике сравнивать текущие координаты с концом отрезка не корректно, так как промах на 1 пиксель из-за округления, например, отправит наш объект в бесконечное путешествие по прямой. Можно, например, прикинуть время, которое потребуется объекту, чтобы дойти до конца отрезка, поделив длину отрезка на скорость объекта. Условием продолжения движения будет время от начала движения T меньшее времени, которое требуется, чтобы пройти отрезок.

Движение по окружности программируется иначе.

Пусть (x0;y0) – центр окружности движения. Тогда координаты объекта в момент времени T от начала движения, движущегося со скоростью 10 шагов в секунду по окружности, равны:

(xc;yc) = (x0; y0) + R \* (cos(V\*T); sin(V\*T)),

где R – радиус окружности.

В этом случае в отличие от предыдущего 10 шагов в секунду – угловая скорость, т.е. измеряется в радиан/сек.

#### Практические задания

***Вариант 1***

Реализовать перемещение по экрану объектов Муравьи-рабочие (символ «O») и объектов Муравьи-воины (символ «@»).

Муравьи-рабочие двигаются в один из углов области их обитания (например, [0;0]) по прямой со скоростью V, а затем возвращаться обратно в точку своего рождения с той же скоростью.

Муравьи-воины двигаются по окружности с радиусом R со скоростью 10 шагов в секунду.

Количество объектов обоих типов задается динамически при запуске программы.

Задача должна быть решена с использованием принципов многопоточности.

***Вариант 2***

Реализовать перемещение по экрану объектов Пчелы-рабочие (символ «O») и объектов Трутни (символ «@»).

Пчелы-рабочие двигаются в один из углов области их обитания (например, [0;0]) по прямой со скоростью V, а затем возвращаться обратно в точку своего рождения с той же скоростью.

Трутни двигаются хаотично со скоростью 10 шагов в секунду.

Хаотичность достигается случайной сменой направления движения раз в N секунд.

Количество объектов обоих типов задается динамически при запуске программы.

Задача должна быть решена с использованием принципов многопоточности.

***Вариант 3***

Реализовать перемещение по экрану объектов «Золотые рыбки» (символ «->»).Золотые рыбки двигаются по оси X от одного края области обитания до другого со скоростью 10 шагов в секунду. Стрелка должна меняться в соответствии с направлением движения.

Количество объектов обоих типов задается динамически при запуске программы.

Задача должна быть решена с использованием принципов многопоточности.

***Вариант 4***

Реализовать перемещение по экрану объектов «Обыкновенные кролики» (символ «O») и объектов «Альбиносы» (символ «@»).

Обыкновенные кролики двигаются хаотично со скоростью 10 шагов в секунду. Хаотичность достигается случайной сменой направления движения раз в N секунд.

Альбиносы двигаются по оси X от одного края области обитания до другого со скоростью 10 шагов в секунду.

Количество объектов обоих типов задается динамически при запуске программы.

Задача должна быть решена с использованием принципов многопоточности.

***Вариант 5***

Реализовать перемещение по экрану объектов «Грузовые машины» (символ «O») и объектов «Легковые машины» (символ «@»).

Грузовые машины двигаются в левую верхнюю четверть области симуляции (т.е. прямоугольник с верхним-левым углом в точке 0;0, шириной/длиной = (w/2;h/2), где w и h – ширина и длина области симуляции) со скоростью 10 шагов в секунду по прямой. Конечная точка движения – случайная точка в пределах этой области. Если машина сгенерировалась сразу в этой области, то она никуда не движется. По прибытии в конечную точку машина больше не движется.

Легковые машины двигаются в нижнюю правую четверть области симуляции (т.е. прямоугольник с верхним-левым углом в точке w/2;h/2, шириной/длиной = (w/2;h/2), где w и h – ширина и длина области симуляции) со скоростью 10 шагов в секунду по прямой. Конечная точка движения – случайная точка в пределах этой области. Если машина сгенерировалась сразу в этой области, то она никуда не движется. По прибытии в конечную точку машина больше не движется.

Количество объектов обоих типов задается динамически при запуске программы.

Задача должна быть решена с использованием принципов многопоточности.

***Вариант 6***

Реализовать перемещение по экрану объектов «Разработчик» (символ «O») и объектов «Менеджер» (символ «@»).

Разработчики двигаются хаотично со скоростью 10 шагов в секунду. Хаотичность достигается случайной сменой направления движения раз в N секунд. Разработчики могут исчезать случайным образом. Если разработчик исчезает, то появляетс такой же, но в другой области экрана (новый поток).

Менеджеры двигаются по окружности с радиусом R со скоростью 10 шагов в секунду.

Количество объектов обоих типов задается динамически при запуске программы.

Задача должна быть решена с использованием принципов многопоточности.

***Вариант 7***

Реализовать перемещение по экрану объектов «Капитальные дома» (символ «O») и объектов «Деревянные дома» (символ «@»).

Капитальные дома двигаются (в городах будущего и не такое возможно) в левую верхнюю четверть области симуляции (т.е. прямоугольник с верхним-левым углом в точке 0;0, шириной/длиной = (w/2;h/2), где w и h – ширина и длина области симуляции) со скоростью 10 шагов в секунду по прямой. Конечная точка движения – случайная точка в пределах этой области. Если дом сгенерировался сразу в этой области, то он никуда не движется. По прибытии в конечную точку дом больше не движется.

Деревянные дома после генерации начинают двигаться в нижнюю правую четверть области симуляции (т.е. прямоугольник с верхним-левым углом в точке w/2;h/2, шириной/длиной = (w/2;h/2), где w и h – ширина и длина области симуляции) со скоростью 10 шагов в секунду по прямой. Конечная точка движения – случайная точка в пределах этой области. Если дом сгенерировался сразу в этой области, то он никуда не движется. По прибытии в конечную точку дом больше не движется.

Количество объектов обоих типов задается динамически при запуске программы.

Задача должна быть решена с использованием принципов многопоточности.

***Вариант 8***

Реализовать перемещение по экрану объектов Муравьи-рабочие (символ «O») и объектов Муравьи-воины (символ «@»).

Автомобили двигаются по оси X от одного края области симуляции до другого со скоростью 10 шагов в секунду.

Мотоциклы двигаются по оси Y от одного края области симуляции до другого со скоростью 10 шагов в секунду.

Достигнув края, автомобиль или мотоциклист исчезают и вместо них появляются другие, но с другой стороны экрана (новый поток).

Количество объектов обоих типов задается динамически при запуске программы.

Задача должна быть решена с использованием принципов многопоточности.

***Вариант 9***

Реализовать перемещение по экрану объектов «Рабочие» (символ «O») и объектов «Менеджеры» (символ «@»).

Рабочие двигаются хаотично со скоростью 10 шагов в секунду в пределах выделенной области экрана. Хаотичность достигается случайной сменой направления движения раз в N секунд. Рабочий с вероятностью 2% исчезнет на следующем шаге.

Менеджеры двигаются по периметру выделенной области экрана со скоростью 10 шагов в секунду.

Количество объектов обоих типов задается динамически при запуске программы.

Задача должна быть решена с использованием принципов многопоточности.

***Вариант 10***

Реализовать перемещение по экрану объектов «Юр. лица» (символ «O») и объектов «Физ. лица» (символ «@»).

Юр. лица двигаются в левую верхнюю четверть области симуляции (т.е. прямоугольник с верхним-левым углом в точке 0;0, шириной/длиной = (w/2;h/2), где w и h – ширина и длина области симуляции) со скоростью 10 шагов в секунду по прямой. Конечная точка движения – случайная точка в пределах этой области. Если запись сгенерировалась сразу в этой области, то она никуда не движется. По прибытии в конечную точку запись больше не движется.

Физ. лица двигаются в нижнюю правую четверть области симуляции (т.е. прямоугольник с верхним-левым углом в точке w/2;h/2, шириной/длиной = (w/2;h/2), где w и h – ширина и длина области симуляции) со скоростью 10 шагов в секунду по прямой. Конечная точка движения – случайная точка в пределах этой области. Если запись сгенерировалась сразу в этой области, то она никуда не движется. По прибытии в конечную точку запись больше не движется.

Количество объектов обоих типов задается динамически при запуске программы.

Задача должна быть решена с использованием принципов многопоточности.