

JAVA

ADVANCED



ЧТО БЫЛО В ПРОШЛЫЙ РАЗ?

- Системы контроля версия
- git
- github, gitverse
- Подход gitflow
- Основы работы с git в Idea
- Вспомнили синтаксис Java
- Вспомнили основы ООП в Java

ЧТО БУДЕТ СЕГОДНЯ?

- Системы сборки проектов
- Apache Maven
 - Репозитории
 - Архетипы
 - Зависимости
 - Сборка Jar-файла
 - Сборка исполняемого Jar-файла
- Еще немного git/github-a
- Пирамида тестирования
- Аннотации
- Вспомним про исключения
- Модульные тесты на JUnit
- Принцип F.I.R.S.T. для юнит-тестов

СБОРКА JAR-ФАЙЛА ВРУЧНУЮ

ИЛИ ЗАЧЕМ НУЖНЫ СИСТЕМЫ СБОРКИ?

КОМПИЛЯЦИЯ ИСХОДНИКОВ

```
javac -sourcepath src -d bin -encoding UTF8 ./src/lesson_10/time_ex/*.java ./src/lesson_10/text_block/*.java
```

где

- `sourcepath` – указывает, где искать исходники для зависимостей в классах
- `d` – показывает, куда сложить class-файлы
- `encoding` – если есть проблемы с кодировкой

СОЗДАНИЕ МАНИФЕСТА

```
cat .\manifest.MF  
Main-Class: lesson_10.time_ex.Application
```

СОЗДАНИЕ JAR-ФАЙЛА

```
C:\Program Files\Java\jdk-17\bin\jar -cmf .\manifest.MF lesson10.jar -C bin .
```

А ЕЩЕ ЕСТЬ ЗАВИСИМОСТИ И ИХ ВЕРСИИ

СИСТЕМЫ СБОРКИ

- в разы облегчают компиляцию, сборку проекта любого размера и сложности
- позволяют управлять зависимостями
- кастомизировать процесс сборки за счет плагинов, задач и прочего в зависимости от системы сборки

ОСНОВНЫЕ СИСТЕМЫ СБОРКИ

APACHE ANT

- старый и почти не встречаемый (к счастью)

```
<project name="MyProject" default="dist" basedir=".">
  <description>
    simple example build file
  </description>
  <!-- set global properties for this build -->
  <property name="src" location="src"/>
  <property name="build" location="build"/>
  <property name="dist" location="dist"/>

  <target name="init">
    <!-- Create the time stamp -->
    <tstamp/>
    <!-- Create the build directory structure used by compile -->
    <mkdir dir="${build}"/>
  </target>

  <target name="compile" depends="init"
    description="compile the source">
    <!-- Compile the Java code from ${src} into ${build} -->
    <javac srcdir="${src}" destdir="${build}"/>
  </target>

  <target name="dist" depends="compile"
    description="generate the distribution">
    <!-- Create the distribution directory -->
    <mkdir dir="${dist}/lib"/>
```

ОСНОВНЫЕ СИСТЕМЫ СБОРКИ

АРАСНЕ MAVEN

- maven - основан на xml, плагинах

```
<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 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>Maven Quick Start Archetype</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <build>
    <pluginManagement>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-compiler-plugin</artifactId>
          <version>3.8.1</version>
        </plugin>
      </plugins>
    </pluginManagement>
  </build>
</project>
```

ОСНОВНЫЕ СИСТЕМЫ СБОРКИ

GRADLE

- gradle - описывает то же, что maven, только на языках groovy или kotlin
- помимо прочего, можно писать свои задачи на языке, соответственно, groovy или kotlin

```
plugins {  
    id 'application'  
}  
  
repositories {  
    mavenCentral()  
}  
  
dependencies {  
    testImplementation libs.junit.jupiter  
    testRuntimeOnly 'org.junit.platform:junit-platform-launcher'  
    implementation libs.guava  
}  
  
java {  
    toolchain {  
        languageVersion = JavaLanguageVersion.of(11)  
    }  
}  
  
application {  
    mainClass = 'org.example.App'  
}  
  
tasks.named('test') {  
    useJUnitPlatform()  
}
```


MAVEN

- Система сборки, основанная на POM-файлах
- POM-файл - это xml-файл, содержащий:
 - информацию о проекте (группа, артефакт, версия, автор и т.д.)
 - зависимости проекта
 - конфигурацию плагинов
 - задачи
 - профили

MAVEN

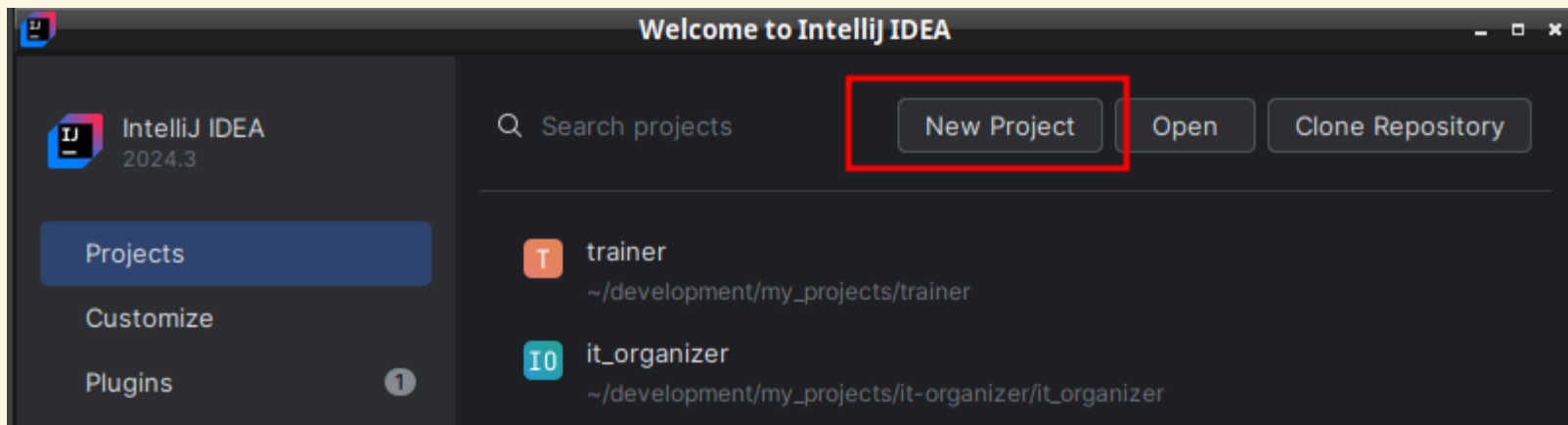
АРХЕТИПЫ

- У maven есть ряд predefined архетипов, которые можно использовать для создания базового pom-файла и структуры папок
 - maven-archetype-quickstart
 - maven-archetype-site
 - maven-archetype-webapp
 - и т.д.
- Можно создавать и свои архетипы

MAVEN

СОЗДАНИЕ В IDE БАЗОВОЙ СТРУКТУРЫ ИЗ АРХЕТИПА

- Архетип можно применять только к пустой папке.
Так что, создадим проект в новой папке и скопируем нужные нам файлы оттуда в наш проект
- Идем в Idea, жмем New Project



MAVEN

СОЗДАНИЕ В IDE БАЗОВОЙ СТРУКТУРЫ ИЗ АРХЕТИПА

- Заполняем поля:
 - Name: `trainer`
 - Location: Где хотим расположить проект
 - Archetype: `maven-archetype-quickstart`
 - Раскрываем Advanced Settings
 - GroupId: `ru.spbu.<ваш_ник_или_фамилия>`
- Жмем Create

The screenshot shows the 'New Project' dialog in IntelliJ IDEA. On the left, under 'Generators', 'Maven Archetype' is selected. The main configuration area on the right has the following fields:

- Name:** `trainer`
- Location:** `~/development/my_projects/trainer`. Below this, it says 'Project will be created in: ~/development/my_projects/trainer/trainer'. There is an unchecked checkbox for 'Create Git repository'.
- JDK:** `temurin-17 Eclipse Temurin 17.0.13`
- Catalog:** `Internal` (with a 'Manage catalogs...' link).
- Archetype:** `.maven.archetypes:maven-archetype-quickstart` (with an 'Add...' button).
- Version:** `1.1`

Below these fields is the 'Additional Properties' section, which is currently empty and shows 'No properties'.

At the bottom, the 'Advanced Settings' section is expanded, showing:

- GroupId:** `org.akhitev` (highlighted with a blue box)
- ArtifactId:** `trainer`

MAVEN

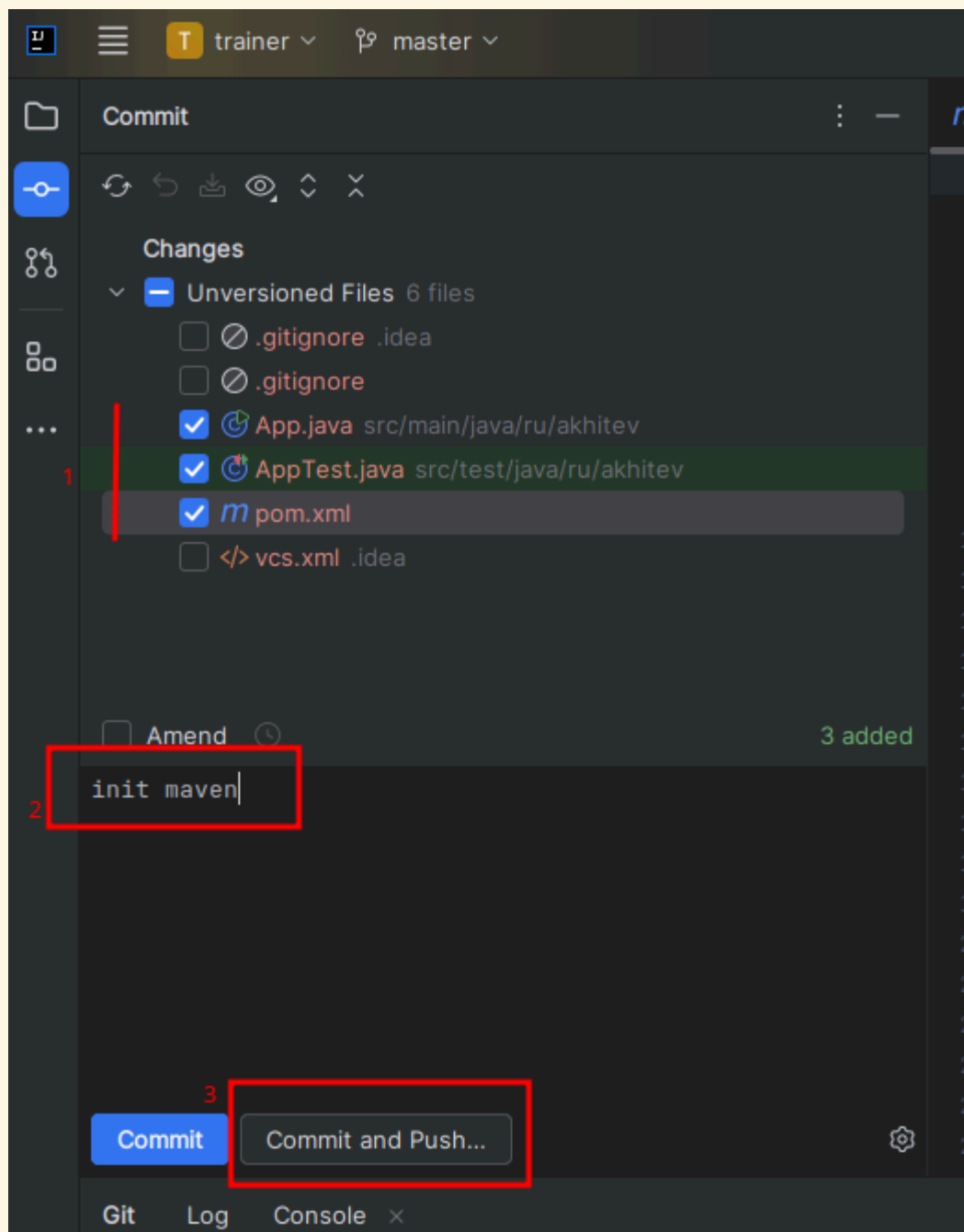
ПЕРЕНОС В НАШ ПРОЕКТ

- Копируем файлы в наш репозиторий
- Закрываем и открываем снова Idea
- Она распознает maven-проект
- Соглашаемся с тем, что нужно загрузить модуль Maven

GIT

КОММИТИМ НАШИ ИЗМЕНЕНИЯ

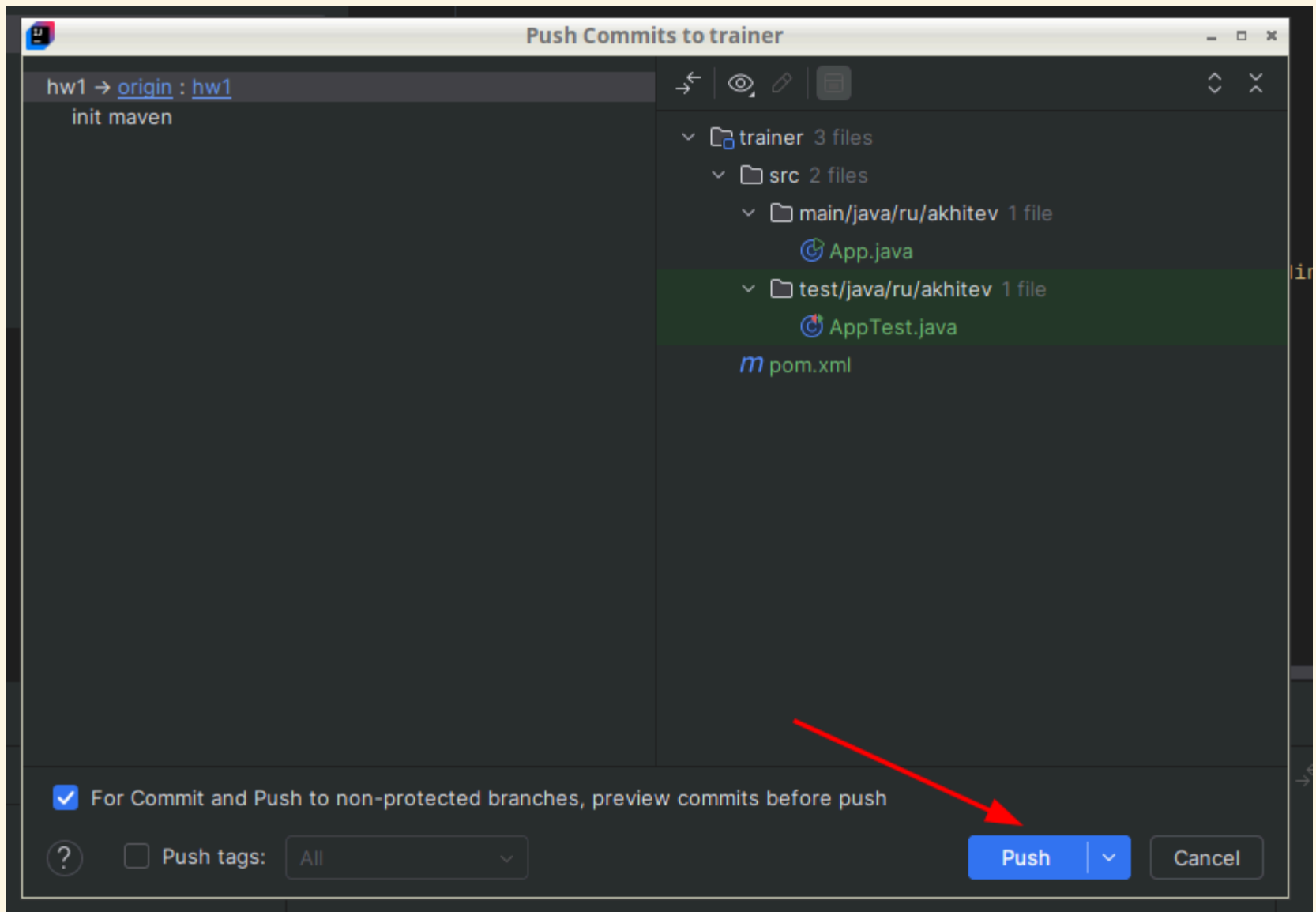
- Выбираем
 - `pom.xml`,
 - `App.java`,
 - `AppTest.java`
- Пишем комментарий (он обязателен)
- Жмем `Commit and Push`



GIT

КОММИТИМ НАШИ ИЗМЕНЕНИЯ

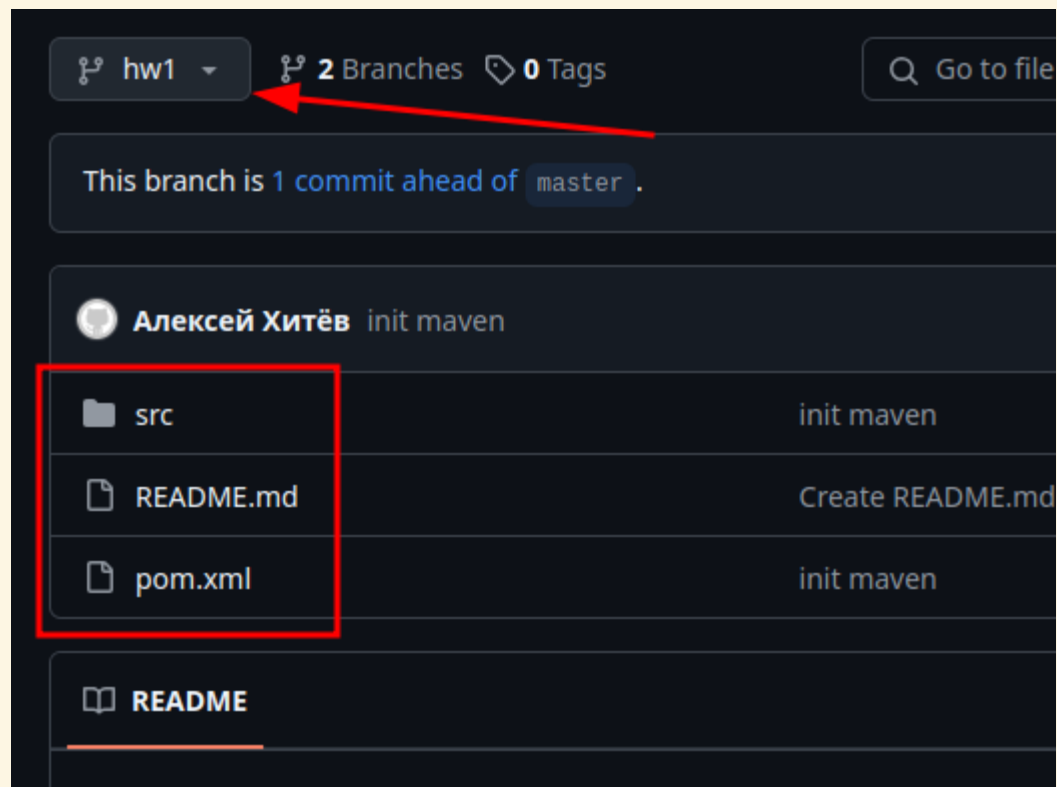
- Проверяем, что не закоммитили ничего лишнего
- Жмем Push



GITHUB

ПРОВЕРЯЕМ, ЧТО ИЗМЕНЕНИЯ ДОЕХАЛИ

- Идем в gitHub
- Переключаемся на нашу ветку hw1
- Проверяем, что изменения приехали



СТРУКТУРА POM-ФАЙЛА

ГРУППА, АРТЕФАКТ, ВЕРСИЯ

- Вернемся к Idea
- Откроем `pom.xml`

```
1 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" >
2   <modelversion>4.0.0</modelversion>
3
4   <groupid>ru.akhitev</groupid>
5   <artifactid>trainer</artifactid>
6   <version>1.0-SNAPSHOT</version>
7   <packaging>jar</packaging>
8
9   ...
10 </project>
```

`pom.xml`

- `groupId` - группа проектов.
Пробелы и двоеточия недопускаются
- `artifactId` - идентификатор самого проекта
Тоже без двоеточий и пробелов
- `version` - версия проекта
- Исходя из этих трех атрибутов, складывается путь к артефакту в репозитории, но об этом чуть позже

СТРУКТУРА POM-ФАЙЛА

ТИП УПАКОВКИ

```
1 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" >
2   <modelversion>4.0.0</modelversion>
3
4   <groupid>ru.akhitev</groupid>
5   <artifactid>trainer</artifactid>
6   <version>1.0-SNAPSHOT</version>
7   <packaging>jar</packaging>
8
9   ...
10 </project>
```

pom.xml

- **packaging** - показывает, как нужно упаковать наш артефакт при сборке
 - **jar** - обычное приложение или библиотека
 - **war** - веб-приложение, предназначенное для разворачивания на сервере приложений типа Apache Tomcat, Wildfly и пр.
 - **ear** - enterprise-приложение, которое разворачивается на Wildfly, WebSphere и других
 - **pom** - подходит для родительских артефактов
- В maven-проектах может быть наследование pom-файлов, создание многомодульных проектов. Но мы об этом подробнее поговорим через занятие

СТРУКТУРА POM-ФАЙЛА

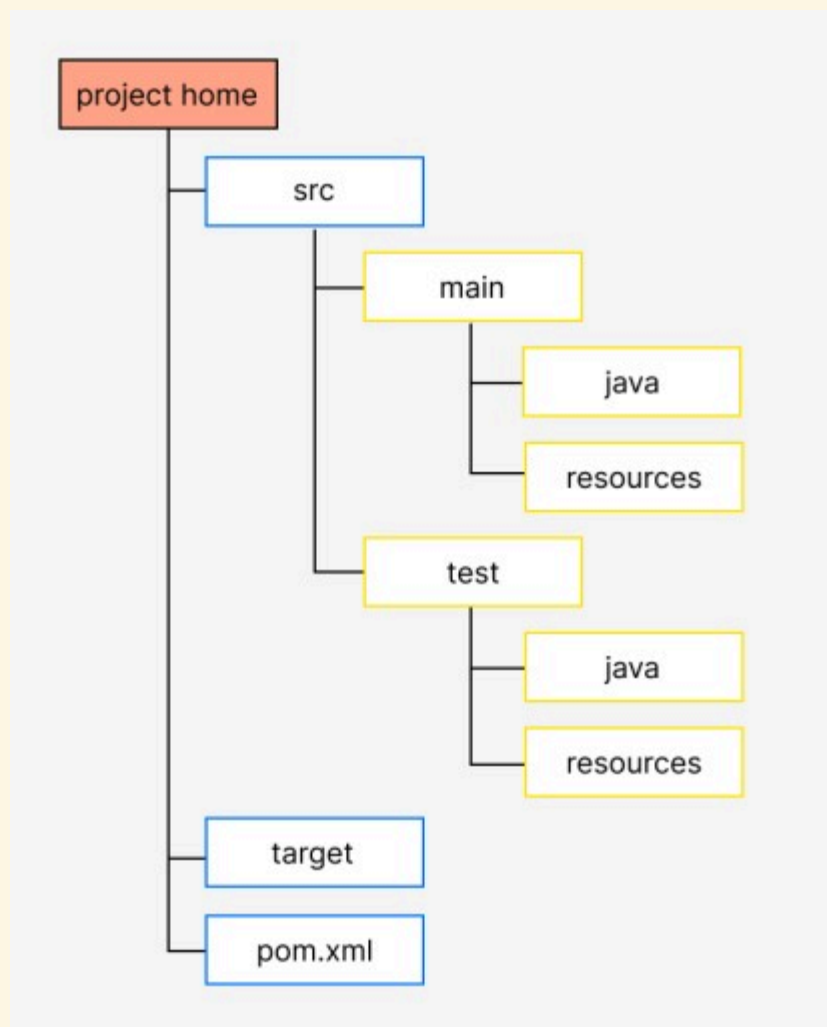
ЗАВИСИМОСТИ

```
1 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" >
2   ...
3
4   <dependencies>
5     <dependency>
6       <groupId>junit</groupId>
7       <artifactId>junit</artifactId>
8       <version>3.8.1</version>
9       <scope>test</scope>
10    </dependency>
11  </dependencies>
12 </project>
```

pom.xml

- Зависимости укладываются в контейнер `dependencies`
- Каждая зависимость представляет собой контейнер `dependency`
- У зависимости есть обязательные атрибуты
 - `groupId`
 - `artifactId`
 - `version`
- Есть и необязательный `scope`, который может принимать значения
 - `compile` - по умолчанию
 - `test` - зависимость подключается только для выполнения тестов и в итоговый артефакт не кладется
 - `provided` - зависимость будет предоставлена в JDK или на том сервере, где будет развернут артефакт. В сборку класть не надо
 - и другие

СТРУКТУРА ПРОЕКТА MAVEN



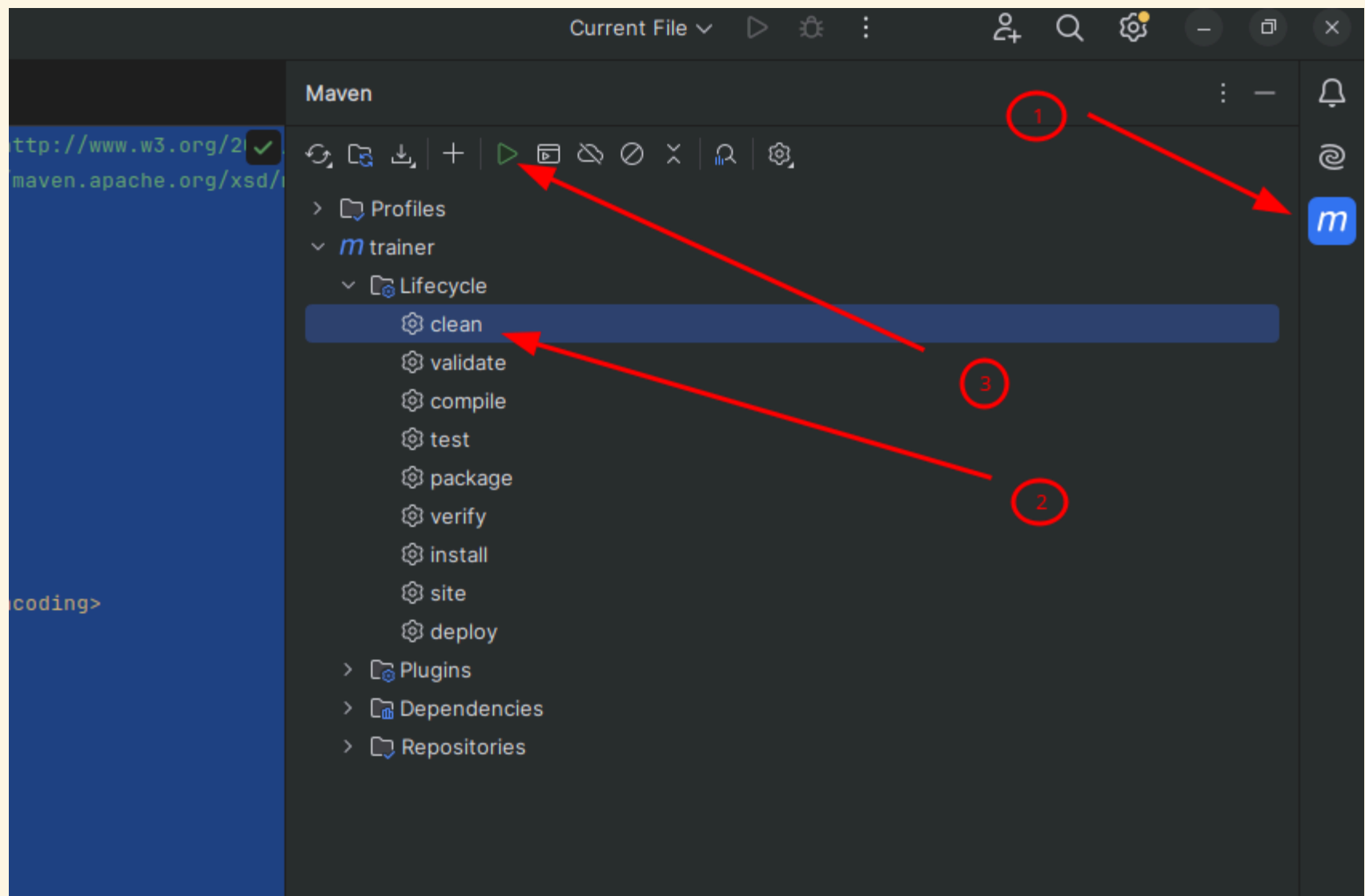
КОМАНДЫ MAVEN

CLEAN

- Удаляет целевой каталог target
- Лучше делать перед каждой сборкой
- Если работаем в консоле, то

```
mvn clean
```

- Если в Idea, то



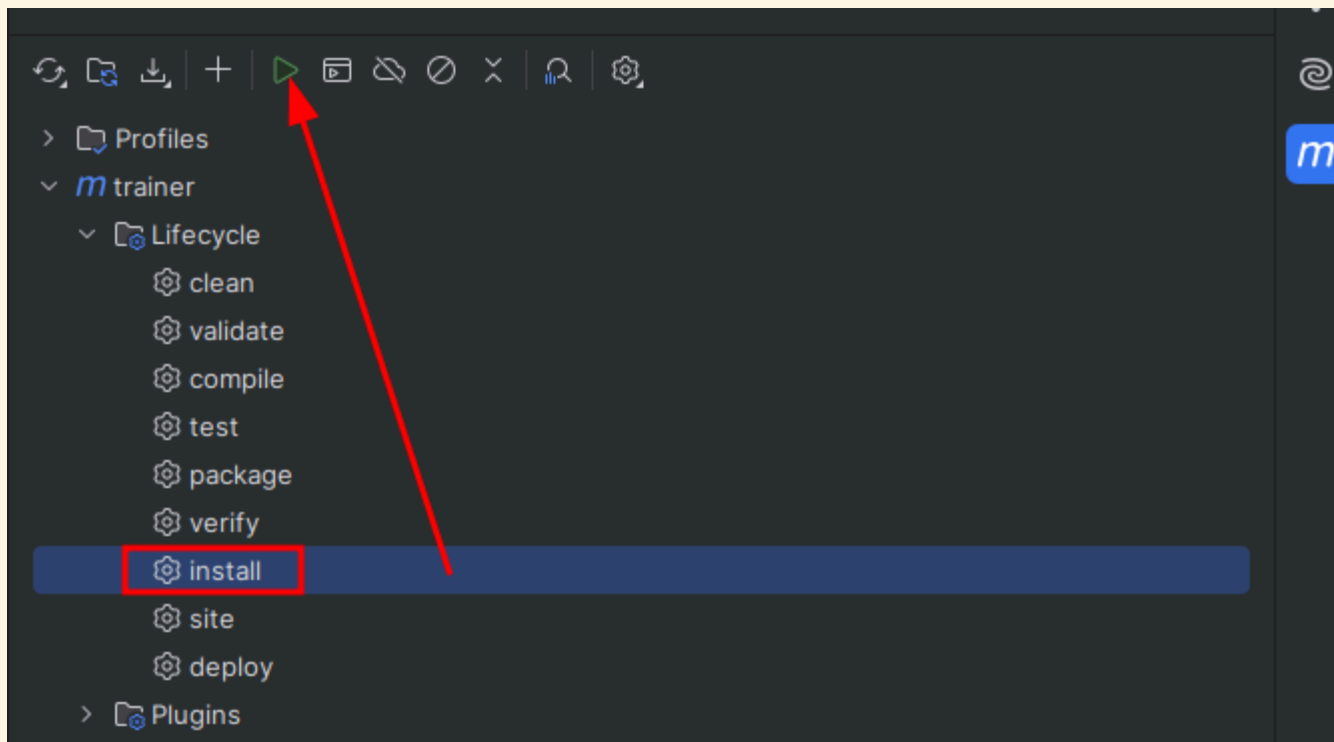
КОМАНДЫ MAVEN

INSTALL

- Собирает проект (получаем jar-, war-, ear- и т.д. файл)
- Кладет полученный артефакт в локальный репозиторий
- Если работаем из консоли, то

```
mvn install
```

- Если работаем в Idea, то



КОМАНДЫ MAVEN

ПРОЧИЕ

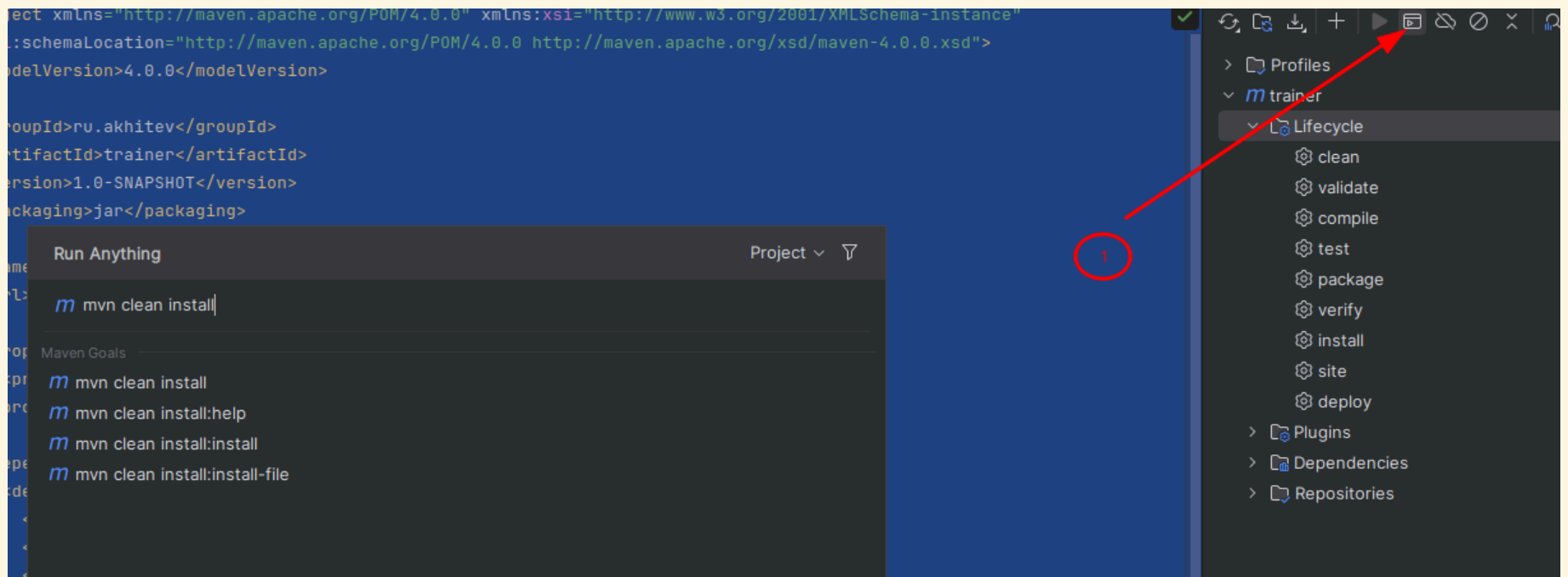
- `compile` - компилирует исходный код
- `package` только собирает проект и создает артефакт
- `validate` - проверяет проект на правильность и что всего хватает для сборки
- и другие

КОМАНДЫ MAVEN

- При работе из консоли, можно запускать сразу несколько команда

```
mvn clean install
```

В Idea тоже можно выполнить несколько команд



ЖИЗНЕННЫЙ ЦИКЛ MAVEN

- Валидация
- Компиляция
- Тестирование -Упаковка
- Интеграционное тестирование
- Установка в локальный репозиторий
- Загрузка в удаленный репозиторий, если он настроен

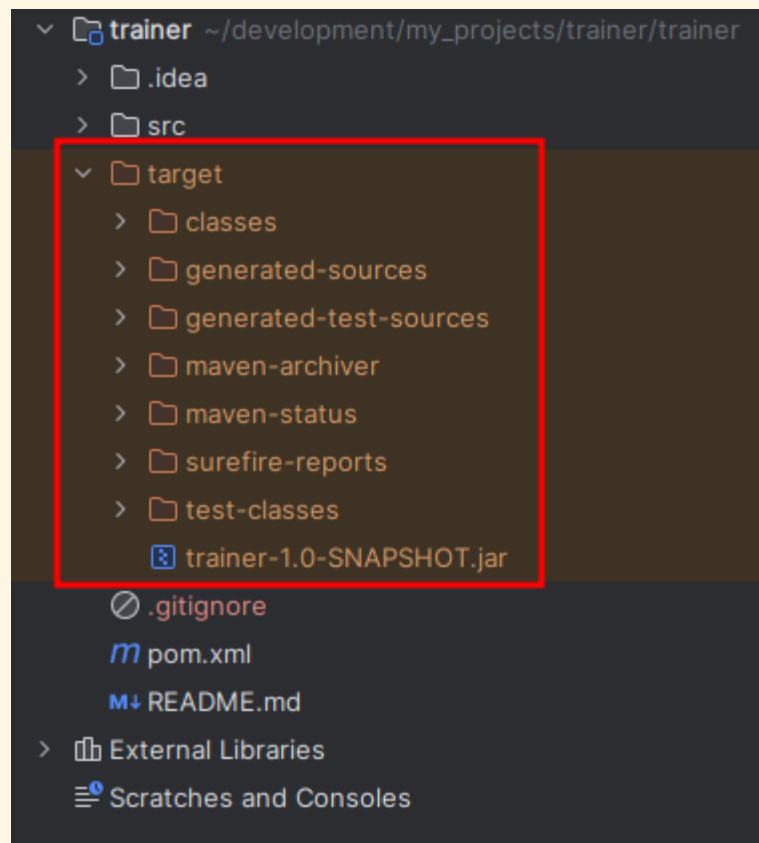
ГДЕ ЛОКАЛЬНЫЙ РЕПОЗИТОРИЙ?

- Сам локальный репозиторий обычно располагается в {домашняя папка пользователя}/.m2/repository
- Конкретно наш собранный проект лежит по пути ~/.m2/repository/ru/akhitev/trainer/1.0-SNAPSHOT
- Как видно, groupId, artefact и version преобразовались в папки в пути к репозиторию
- В папке .m2 еще лежит файл settings.xml. Там описываются данные для доступа к удаленным репозиториям

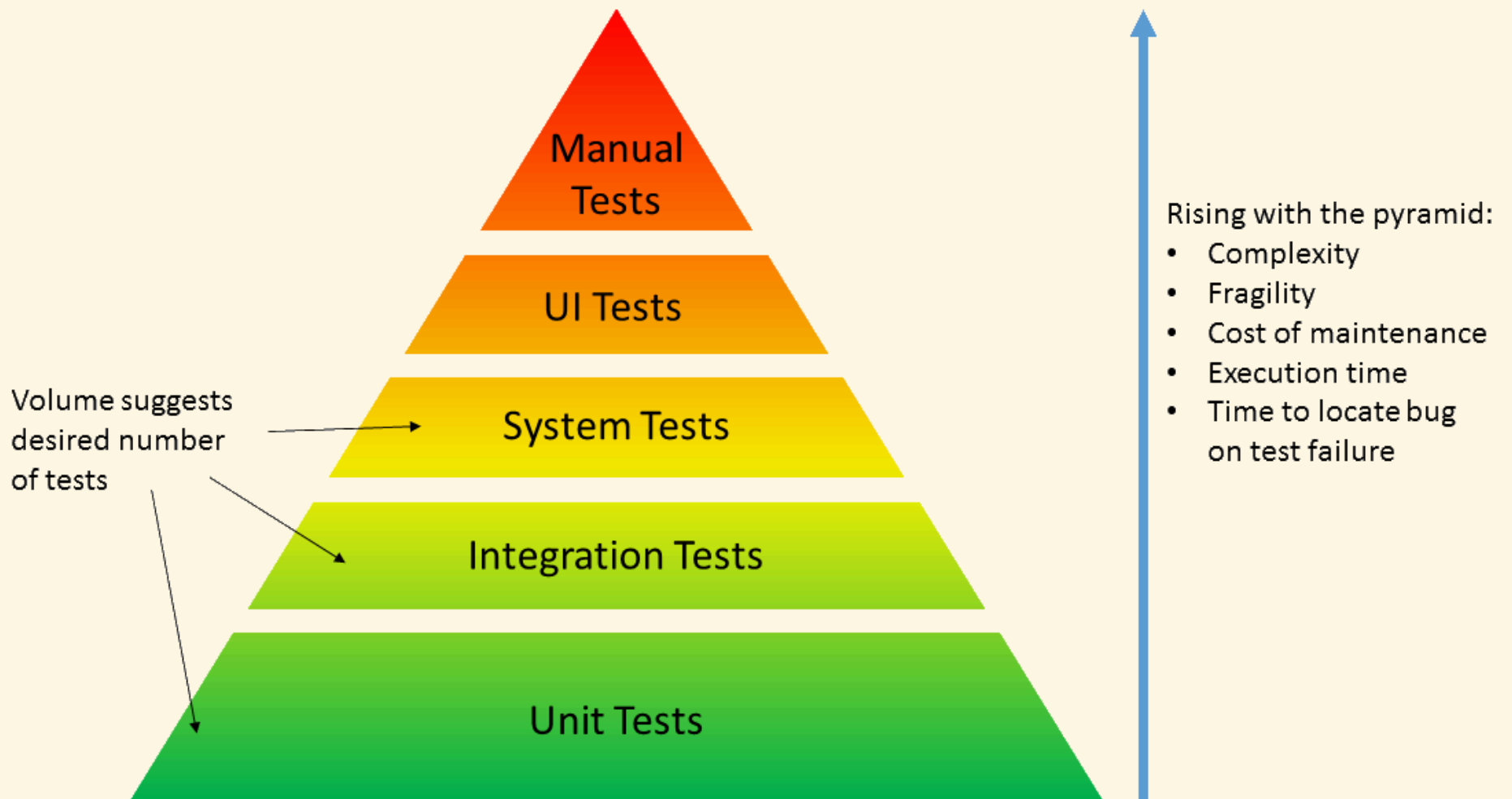
УДАЛЕННЫЕ РЕПОЗИТОРИИ

- Есть общедоступный репозиторий, где лежит куча библиотек
<https://repo.maven.apache.org/maven2/>
- Удобнее пользоваться сайтом
<https://mvnrepository.com/>
- У предприятий обычно свои, куда команды разработки кладут свои артефакты, недоступные снаружи

А ЧТО ЗА ЦЕЛЕВОЙ КАТАЛОГ?



ПИРАМИДА ТЕСТИРОВАНИЯ



ПИРАМИДА ТЕСТИРОВАНИЯ

ЮНИТ-ТЕСТЫ

- Пишутся разработчиками
- Выполняются быстрее всех остальных
- Их больше, чем всех остальных
- При написании юнит-теста, убираю всяческие зависимости от других классов. В этом нам помогают разные заглушки
- Выполняются при каждой сборке, чтоб как можно раньше узнать, что сломали работающий раньше код
- В Java используется библиотека JUnit

ПИРАМИДА ТЕСТИРОВАНИЯ

ИНТЕГРАЦИОННЫЕ, СИСТЕМНЫЕ ТЕСТЫ

- Тоже пишутся разработчиками
- Проверяется работа группы компонентов
- К примеру, от контроллера, принимающего запросы по http до базы данных
- Часто используются test-контейнеры
- Гораздо медленнее, чем юнит-тесты из-за подготовки тестовой БД, компонентов системы
- Их меньше, чем юнит-тестов
- Обычно, тоже выполняются при каждой сборке

ПИРАМИДА ТЕСТИРОВАНИЯ

UI-ТЕСТЫ

- Пишутся автотестирующими
- Используются Selenium/Selenide
- Их еще меньше, чем интеграционных/системных
- Обычно, запускаются отдельным pipeline-ом

QUALITY GATES

- Юнит, интеграционные и системные тесты запускаются при каждой сборке
- Автотесты запускаются при передаче в тестирование и перед подготовкой к релизу
- Smoke-тестирование проводится при поставке новой версии на QA-стенд и после релиза на продуктиве
- Функциональное тестирование проводится при поставке решенной задачи на стенд и перед релизом
- Регрессионное ручное тестирование проводится перед релизом

JUNIT5

- Обновленный JUnit, содержащий необходимые инструменты для создания тестов
- Свежую версию можно посмотреть на сайте

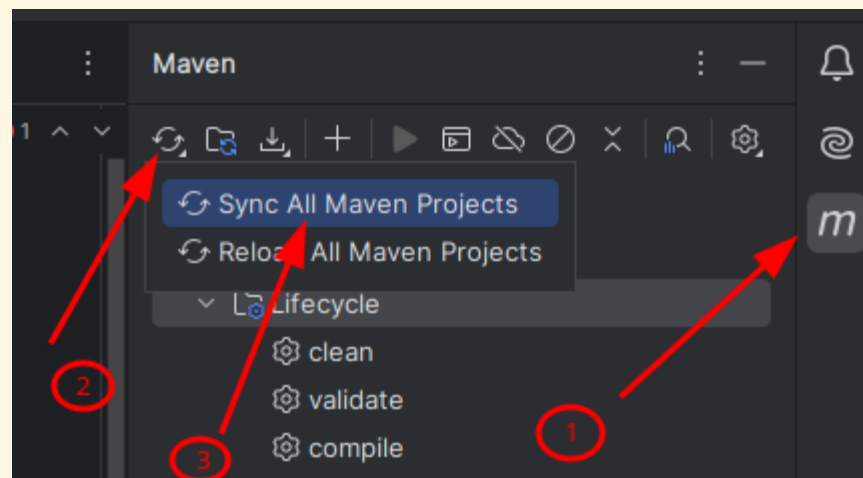
<https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-engine>

- В pom.xml убираем зависимость junit3 и вставляем зависимость на junit5

```
<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>
  ...
  <dependencies>
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-engine</artifactId>
      <version>5.12.0-RC1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

pom.xml

- Чтобы зависимость “проросла” в проекте, нужно выполнить синхронизацию проекта



JUNIT5

НА ЧТО ПОХОЖИ ТЕСТЫ?

```
public class TaskTest {
    private Task task;

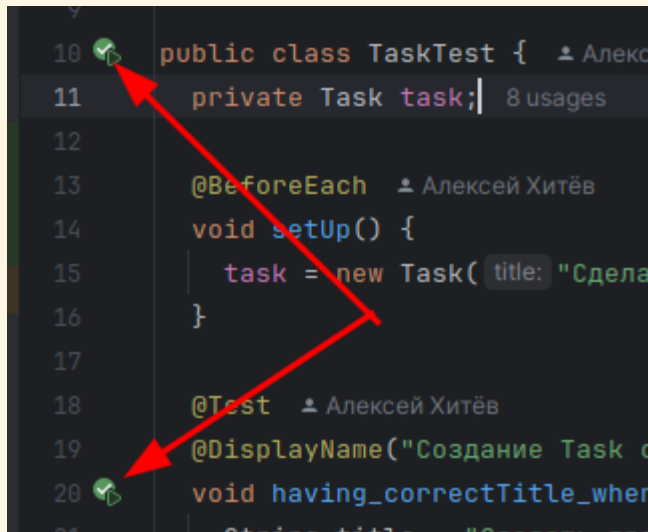
    @BeforeEach
    void setUp() {
        task = new Task("Сделать презентацию для 3 урока");
    }

    @Test
    @DisplayName("Создание Task с корректным title проходит успешно")
    void having_correctTitle_when_newTask_then_created() {
        String title = "Сделать презентацию для 3 урока";
        Task task = new Task(title);
        Assertions.assertEquals(title, task.getTitle());
    }
    ...
}
```

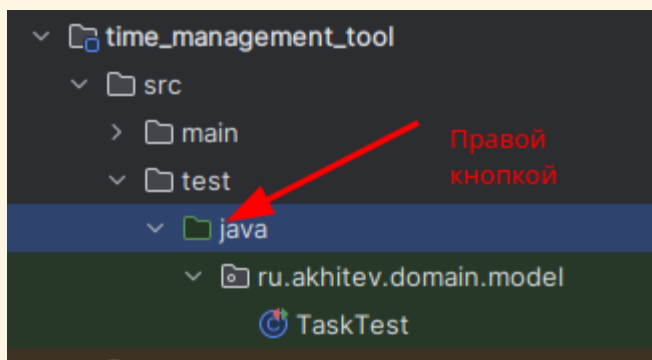
ru.akhitev.domain.model.TaskTest

ЗАПУСК ТЕСТОВ В IDEA

- В Idea для выполнения конкретного теста или тестового класса нужно
 - нажать на иконке запуска рядом с классом или тест-методом



- Чтобы прогнать все тесты, нужно
 - щелкнуть правой кнопкой на папке java в папке test
 - открыть раздел More Run/Debug Options
 - нажать на иконку Run All Tests



АННОТАЦИИ В JAVA

- Аннотации предоставляют информацию, которая необходима для полного описания программы, но не может быть удобно выражена в Java.
- Таким образом, аннотации позволяют сохранить дополнительную информацию о программе в формате, который может быть проверен компилятором.
- Аннотации избавляют разработчика от написания *шаблонного* кода.
- При помощи аннотаций можно держать метаданные в исходном коде Java и пользоваться преимуществами
 - более понятного кода,
 - проверки типов во время компиляции
 - API, упрощающего построение средств обработки аннотаций
- Аннотации могут
 - содержать параметры
 - Применяться к классам, конструкторам, методам, полям и т.д.
 - Быть доступными на разных этапах жизненного цикла (только в коде, при выполнении, только на момент компиляции)

ПРИМЕР АННОТАЦИИ

```
@Target({ElementType.METHOD, ElementType.TYPE})  
@Retention(RetentionPolicy.SOURCE)  
public @interface ShouldBeRefactored {}
```

```
@ShouldBeRefactored  
public void someMethod() {  
    System.out.println("something");  
}
```

АННОТАЦИИ JUNIT5

@TEST

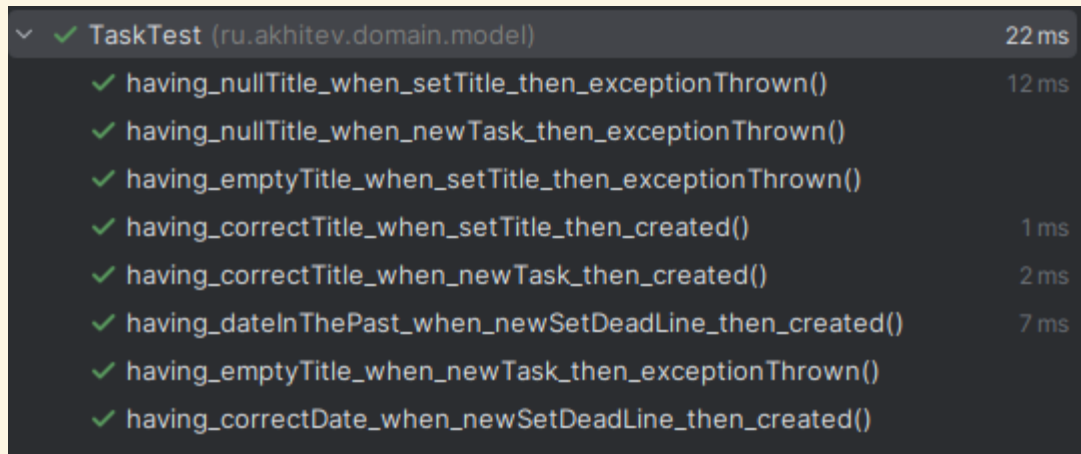
- Аннотация, которая помечает метод, как юнит-тест
- На фазе прогона тестов, методы, помеченные этой аннотацией будут выполнены
- При прогоне тестов с расчетом покрытия тестами, также будут выполняться тесты, помеченные этой аннотацией
- Чтоб ее использовать, нужно не забыть ее импортировать

```
import org.junit.jupiter.api.Test;
```

АННОТАЦИИ JUNIT5

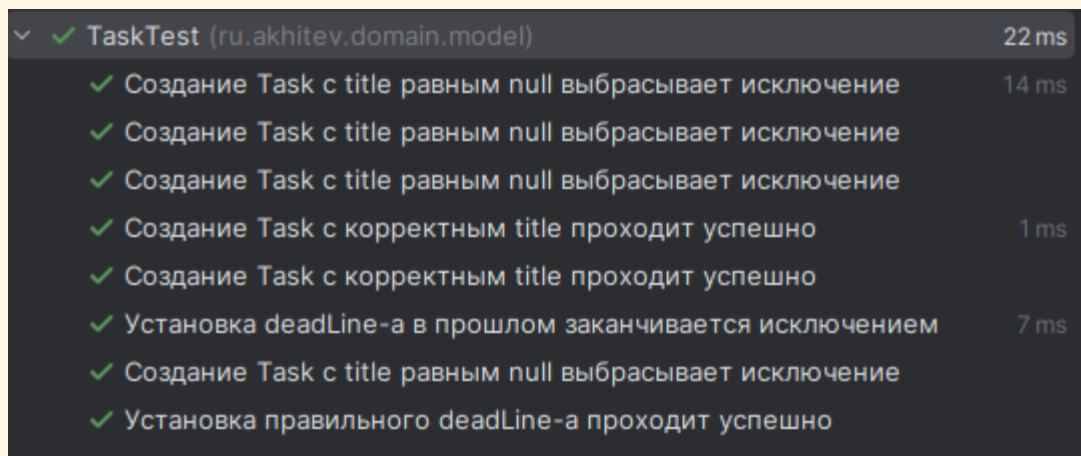
@DISPLAYNAME("...")

- Позволяет задать удобочитаемое название теста для отчетов
- Без этой аннотации



✓ TaskTest (ru.akhitev.domain.model)	22 ms
✓ having_nullTitle_when_setTitle_then_exceptionThrown()	12 ms
✓ having_nullTitle_when_newTask_then_exceptionThrown()	
✓ having_emptyTitle_when_setTitle_then_exceptionThrown()	
✓ having_correctTitle_when_setTitle_then_created()	1 ms
✓ having_correctTitle_when_newTask_then_created()	2 ms
✓ having_dateInThePast_when_newSetDeadLine_then_created()	7 ms
✓ having_emptyTitle_when_newTask_then_exceptionThrown()	
✓ having_correctDate_when_newSetDeadLine_then_created()	

- Используя аннотацию @DisplayName



✓ TaskTest (ru.akhitev.domain.model)	22 ms
✓ Создание Task с title равным null выбрасывает исключение	14 ms
✓ Создание Task с title равным null выбрасывает исключение	
✓ Создание Task с title равным null выбрасывает исключение	
✓ Создание Task с корректным title проходит успешно	1 ms
✓ Создание Task с корректным title проходит успешно	
✓ Установка deadLine-а в прошлом заканчивается исключением	7 ms
✓ Создание Task с title равным null выбрасывает исключение	
✓ Установка правильного deadLine-а проходит успешно	

- Так же не забываем импортировать

```
import org.junit.jupiter.api.DisplayName;
```

АННОТАЦИИ JUNIT5

@BEFOREEACH

- Аннотация, размещается над методом, в котором мы хотим задавать какое то начальное состояние для тестов
- Этот метод будет выполняться перед запуском каждого теста

```
public class TaskTest {  
    private Task task;  
  
    @BeforeEach  
    void setUp() {  
        task = new Task("Сделать презентацию для 3 урока");  
    }  
    ...  
    @Test  
    @DisplayName("Создание Task с корректным title проходит успешно")  
    void having_correctTitle_when_setTitle_then_created() {  
        String title = "Сделать презентацию для 4 урока";  
        task.setTitle(title);  
        Assertions.assertEquals(title, task.getTitle());  
    }  
    ...  
}
```

ru.akhitev.domain.model.TaskTest

ПРОВЕРКИ JUNIT5

ASSERTIONS

- За реализацию самих проверок в тестах отвечает класс `Assertions`
- Он содержит массу методов, которые покрывают основные потребности в юнит-тестах
 - `assertTrue(condition)`
 - `assertFalse(condition)`
 - `assertNull(actual)`
 - `assertNotNull(actual)`
 - `assertEquals(expected, actual)`
 - `assertNotEquals(unexpected, actual)`
 - `assertArrayEquals(expected, actual)`
 - и т.д.
- В этих методах, обычно
 - первым идет корректное значение, с которым мы сравниваем,
 - а уже потом наше значение, которое сравниваем
- Для использования импортируем

```
import org.junit.jupiter.api.Assertions;
```

ПРОВЕРКИ JUNIT5

ASSERTIONS.ASSERTEQUALS

```
1 public class TaskTest {
2     private Task task;
3
4     @BeforeEach
5     void setUp() {
6         task = new Task("Сделать презентацию для 3 урока");
7     }
8
9     @Test
10    @DisplayName("Создание Task с корректным title проходит успешно")
11    void having_correctTitle_when_newTask_then_created() {
12        String title = "Сделать презентацию для 3 урока";
13        Task task = new Task(title);
14        Assertions.assertEquals(title, task.getTitle());
15    }
16    ...
17 }
```

ru.akhitev.domain.model.TaskTest

ПРОВЕРКИ JUNIT5

ASSERTIONS.ASSERTTHROWSEXACTLY

- Проверяет, что было выброшено нужное исключение
- Вначале идет класс нужного исключения
- вторым аргументом идет лямбда, куда мы записываем то действие, которое должно вызвать исключение

```
1 public class TaskTest {
2     private Task task;
3
4     @BeforeEach
5     void setUp() {
6         task = new Task("Сделать презентацию для 3 урока");
7     }
8
9     @Test
10    @DisplayName("Установка deadLine-a в прошлом заканчивается исключением")
11    void having_dateInThePast_when_newSetDeadLine_then_created() {
12        ZonedDateTime deadLine = ZonedDateTime.now().minusDays(1);
13        Assertions.assertThrowsExactly(IllegalArgumentException.class, () -> task.setDeadLine(deadLine));
14    }
15    ...
16 }
```

ru.akhitev.domain.model.TaskTest

ПРИНЦИП F.I.R.S.T

FIRST properties of unit tests

FAST

Many hundreds or thousands per second

Isolates

Failure reasons become obvious

Repeatable

Run repeatedly in any order, any time

Self-validating

No manual evaluation required

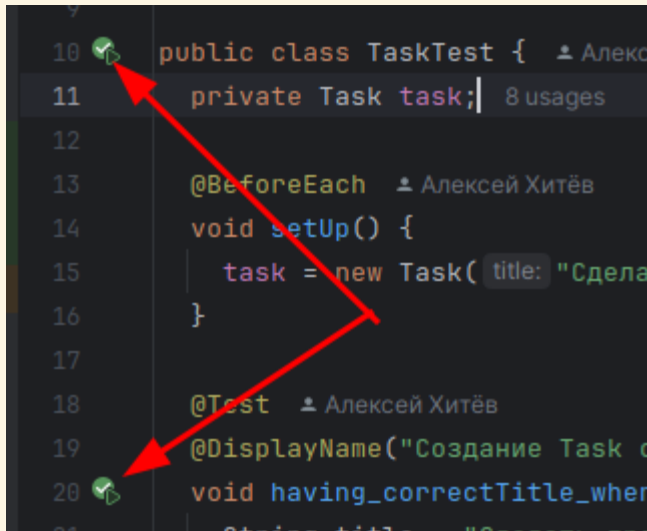
Timely

Written before the code

CODE COVERAGE

ПРОГОН НА ПОКРЫТИЕ ТЕСТАМИ В IDEA

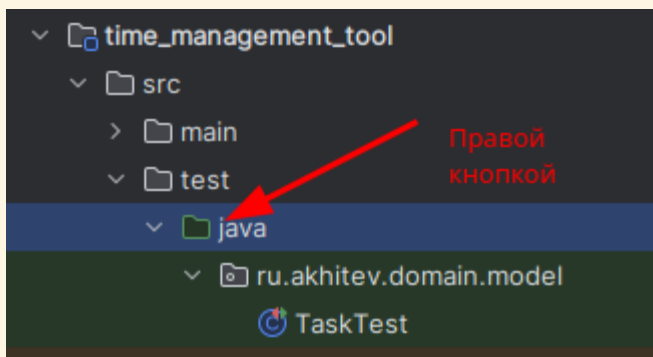
- Покрытие тестами строится исходя из
 - строк кода, которые участвовали при вызове тестов
 - условия, которые выполнялись при работе тестов
- В Idea для выполнения конкретного теста или тестового класса с покрытием нужно
 - нажать правой кнопкой на иконке запуска рядом с классом или тест-методом



- нажать на иконку Run ... with Coverage



- Чтобы прогнать все тесты с покрытием, нужно
 - щелкнуть правой кнопкой на папке `java` в папке `test`
 - открыть раздел More Run/Debug Options
 - нажать на иконку Run ... with Coverage



ВСПОМНИМ ПРО ИСКЛЮЧЕНИЯ

ЧТО ЭТО?

- В случае возникновения ошибок или непредвиденных ситуаций, в Java выбрасывается исключение
- Исключение содержит в себе:
 - Сообщение о проблеме
 - Причину
 - `stackTrace`

ВСПОМНИМ ПРО ИСКЛЮЧЕНИЯ

ПЕРЕХВАТ

- Чтобы поймать ошибку и обработать, в Java используется конструкция
 - `try-catch`
 - `try-catch-finally`
 - `try-finally`
 - `try-with-resources` (с Java 7)
- В блоке `try` завернуто опасное место
- В блоке `catch` задается обработка исключения
- Блок `finally` выполнится в любом случае
- В `try-with-resources` передается объект, реализующий интерфейс `Autoclosable`
 - Java самой закрывает соединение в конце работы и обработать возникающие исключения

ВСПОМНИМ ПРО ИСКЛЮЧЕНИЯ

ВЫБРАСЫВАНИЕ ИСКЛЮЧЕНИЯ

```
private static double calcNetSalary(double grossSalary) {  
    if (grossSalary <= 0) {  
        throw new IllegalArgumentException("Зарплата gross не может быть отрицательной");  
    }  
    return grossSalary * 0.87;  
}
```

ПЕРЕХВАТ ИСКЛЮЧЕНИЯ

```
try {  
    System.out.println(20 / 0);  
} catch (ArithmeticException e) {  
    System.out.println("Произошла арифметическая ошибка: "  
        + e.getMessage() + "\n"  
        + Arrays.toString(e.getStackTrace()));  
}
```

TRY-WITH-RESOURCES

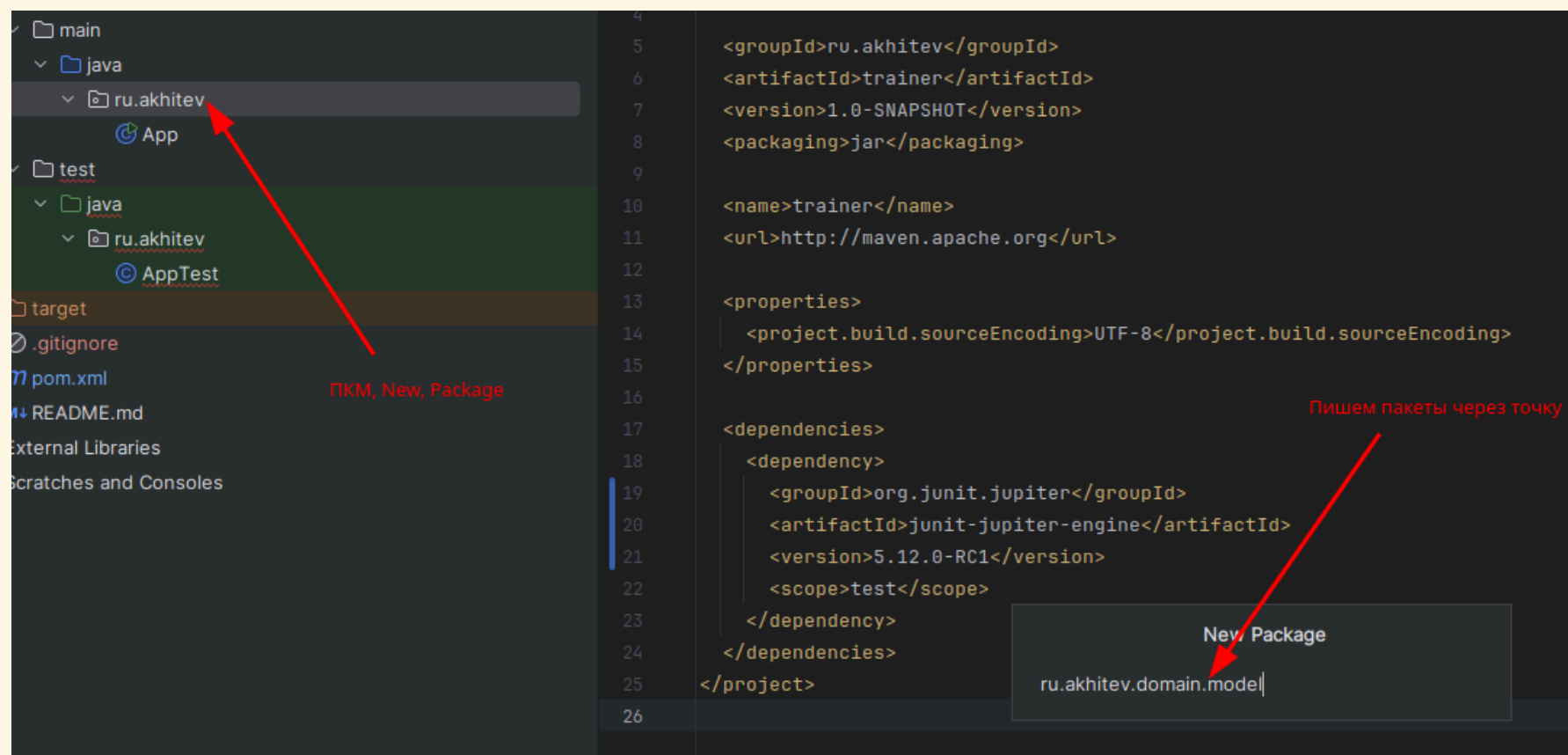
```
try (InputStream inputStream = new ByteArrayInputStream(inputData)) {  
    while(inputStream.available() > 0) {  
        System.out.println(inputStream.read());  
    }  
} catch (IOException e) {  
    e.printStackTrace();  
}
```

ЗАДАНИЕ 2.1

СОЗДАНИЕ КЛАССА МОДЕЛИ OPENQUESTIONCARD В НУЖНОМ ПАКЕТЕ

- Добавить пакет `domain`, а внутри него пакет `model` так, чтобы получилось `ru.<ваш_ник>.domain.model`

Для этого нужно нажать правой кнопкой на пакете `ru.<ваш_ник>` и выбрать `New, Package`

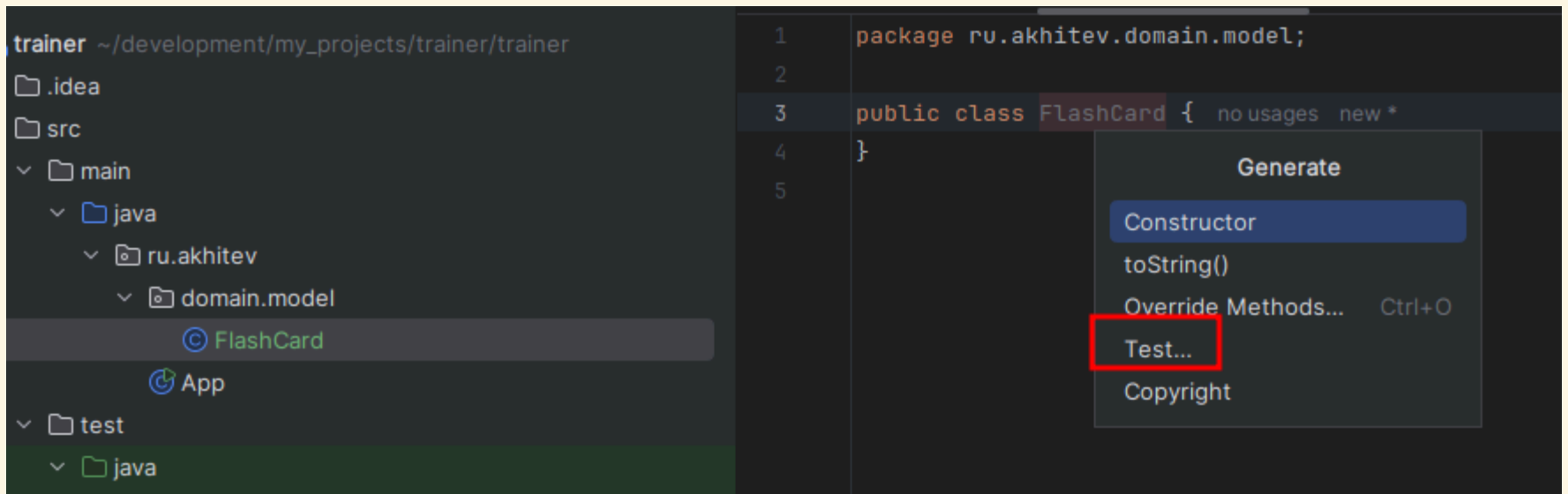


- В новом пакете создать класс `OpenQuestionCard`

ЗАДАНИЕ 2.1

ДЕЛАЕМ КЛАСС ДЛЯ UNIT-ТЕСТОВ

- Идем в `src/test` и удаляем там сгенерированный класс `AppTest`
- Открываем наш класс `OpenQuestionCard` и жмем правой кнопкой на названии класса в редакторе кода
- В появившемся меню выбираем `Generate`
- Выбираем `Test`



- В появившемся окне проверяем, что стоит `JUnit5` и жмем `Ok`
- Проверяем, что класс сгенерировался в `src/test/ru....domain.model`

ЗАДАНИЕ 2.1

НАЧНЕМ ВСЕ ЛОМАТЬ!

- На каждый кейс пишем отдельный unit-тест!
- Если ответ будет непраильным, метод `checkAnswer` вернет `false`?
- Что будет, если ответ, который мы передадим в `checkAnswer` будет `null`?
- А если `question` или `expectedAnswer`, передаваемые в конструктор `null` что произойдет?
А что должно быть по бизнес логике?
- В качестве примера, можно воспользоваться [Unit-тестами](#) к [примеру](#)

ЗАДАНИЕ 2.1

ЗАКОММИТИМ ИЗМЕНЕНИЯ

- Проверяем, что работаем в ветке hw1
- Выбираем файлы в коммит
- Пишем комментарий
- Жмем Commit And Push
- Проверяем GitHub
- Если еще не сделан Pull Request, делаем
- Присылаем мне ссылку на PR

ЗАДАНИЕ 2.1

Полный текст задания с критериями приемки лежит в репозитории

https://github.com/aleksei-khitev/java_adv_22b01_02_e/blob/main/tasks/task_2_1.pdf

В СЛЕДУЮЩЕЙ СЕРИИ

- Слоеная архитектура приложений
- Основные идеи чистого кода
- начало знакомство со Spring

ОБЩИЕ РЕСУРСЫ

- [Таблица с прогрессом](#)
Пароль: student25
- [Группа в Telegram](#)
- Репозиторий с материалами
https://github.com/aleksei-khitev/java_adv_22b01_02_e

ПОЛЕЗНЫЕ МАТЕРИАЛЫ

- [Официальный сайт: Maven за 5 минут](#)
- [Официальный сайт: Архетипы Maven](#)
- [Подробно расписано про scope-ы зависимостей maven](#)
- [Официальный сайт: Жизненный цикл](#)
- [Статья про пирамиду тестирования](#)
- [Официальный гайд для Junit](#)
- [Статья про F.I.R.S.T.](#)

