JAVA

ADVANCED

2

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-а
- Пирамида тестирования
- 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

APACHE 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>
```

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

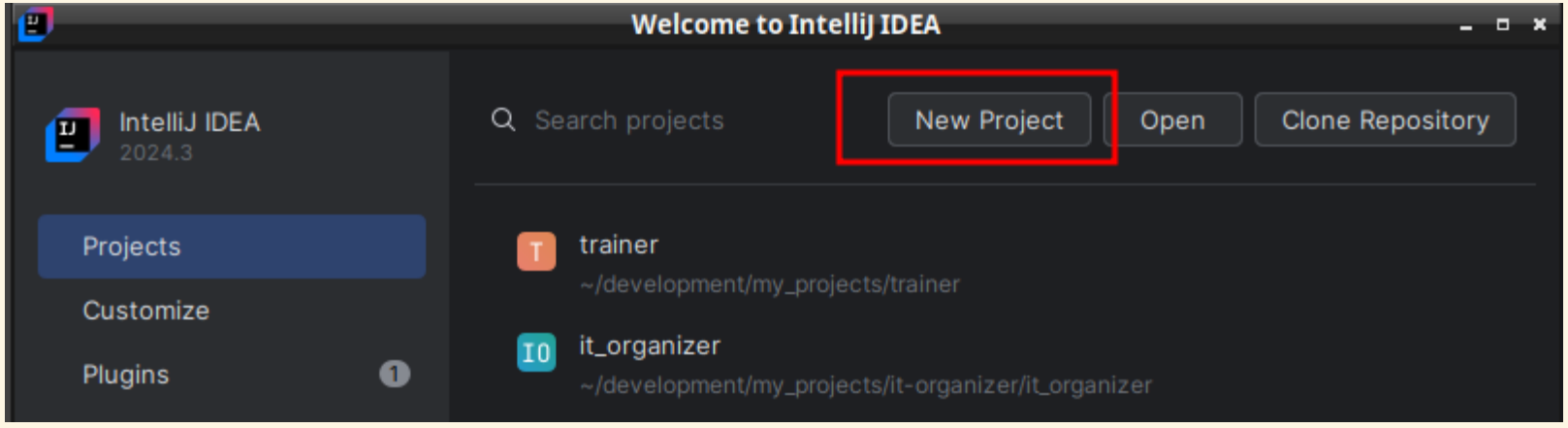
АРХЕТИПЫ

- У maven есть ряд предопределенных 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

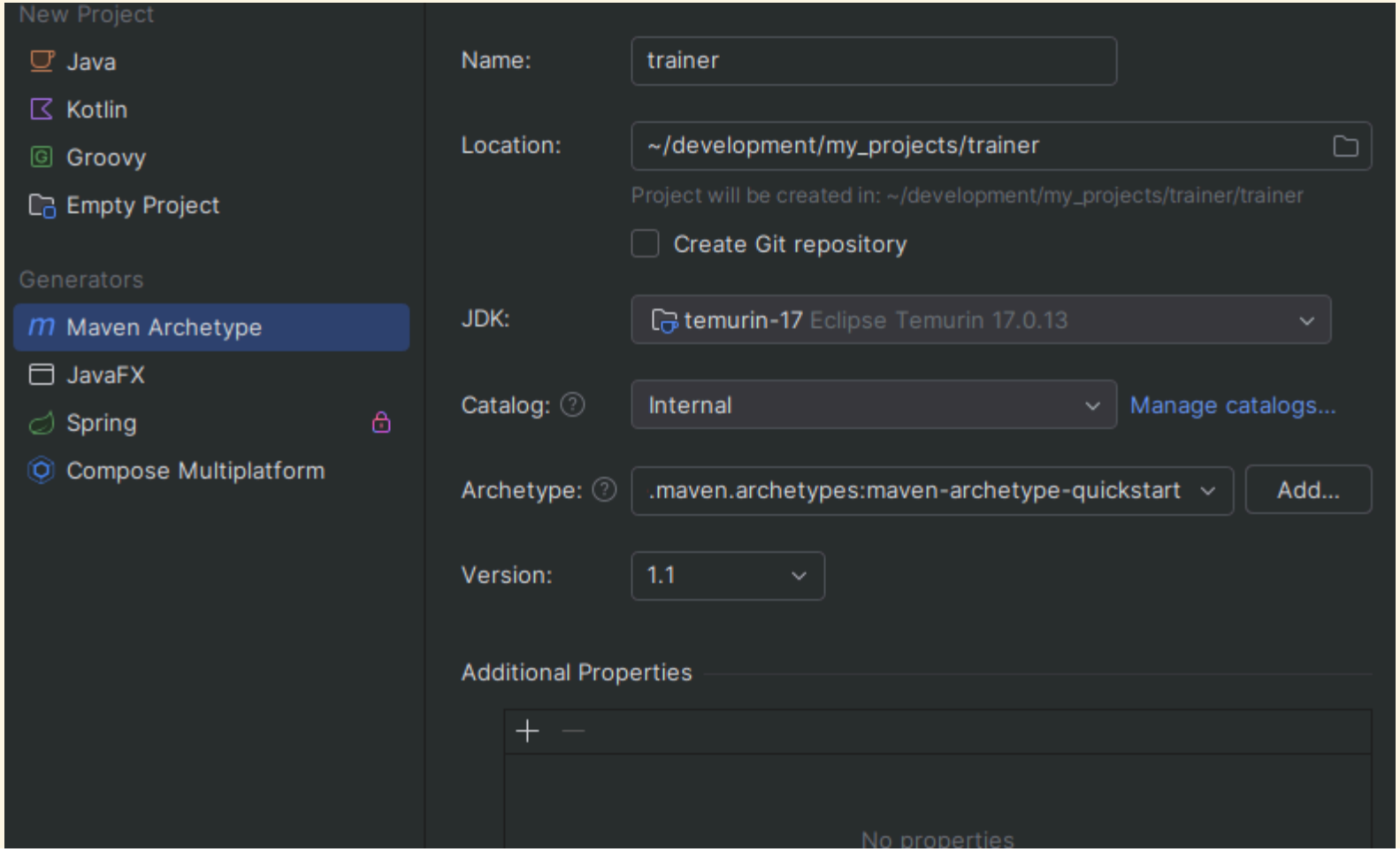
- Архетип можно применять только к пустой папке.
  Так что, создадим проект в новой папке и скопируем нужные нам файлы оттуда в наш проект
- Идем в Idea, жмем `New Project`



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`



MAVEN

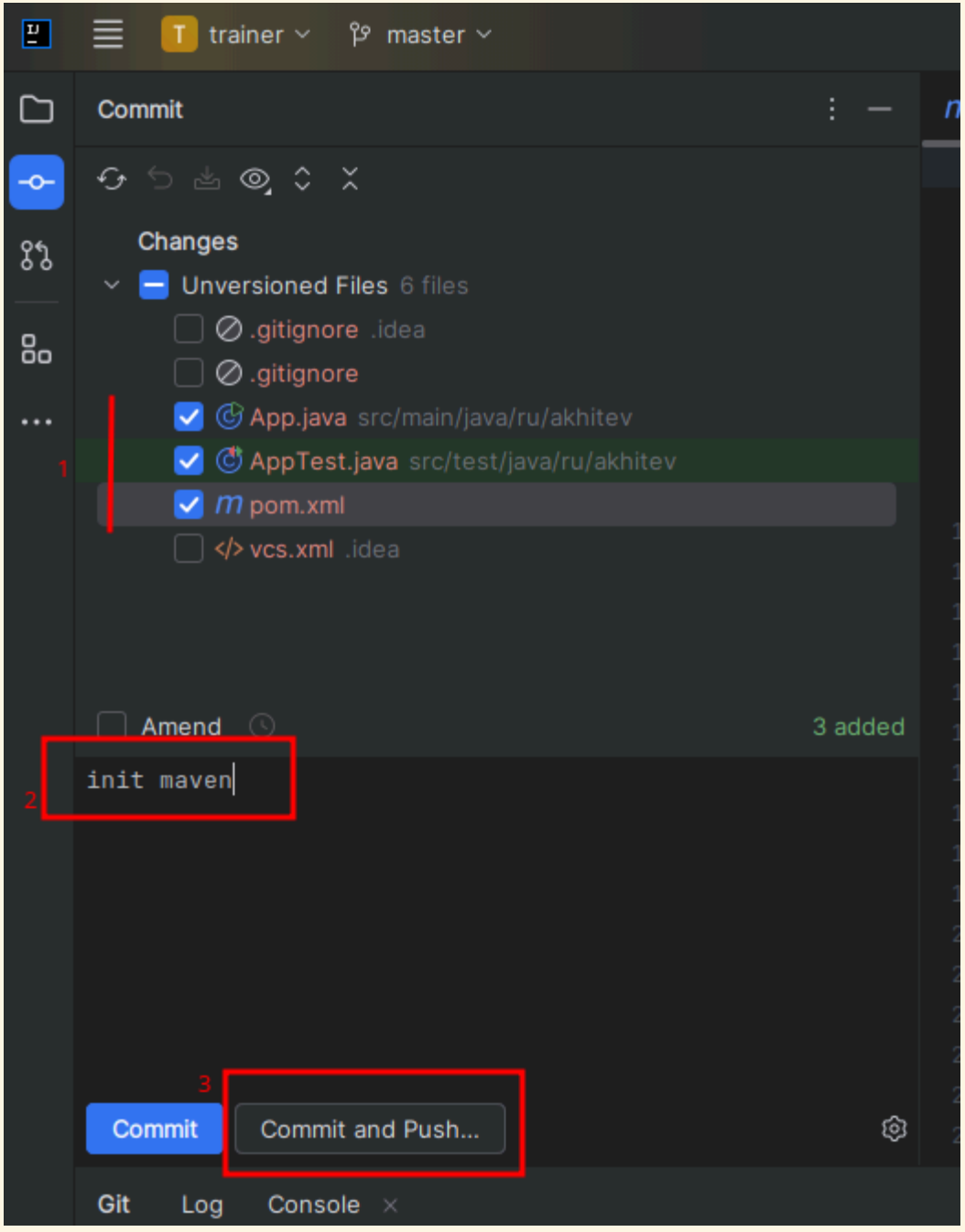
ПЕРЕНОС В НАШ ПРОЕКТ

- 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

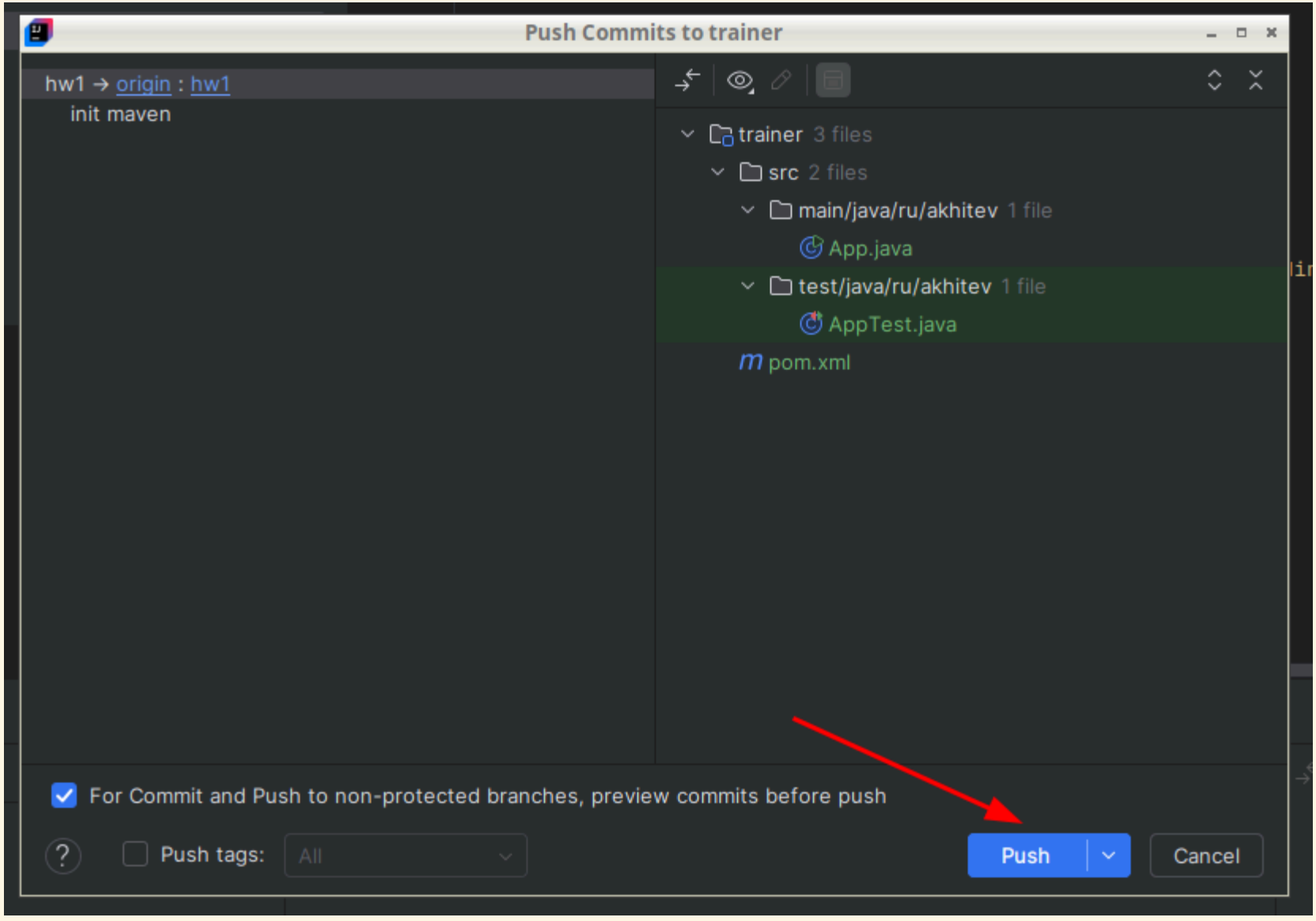
- Выбираем
  - `pom.xml`,
  - `App.java`,
  - `AppTest.java`
- Пишем комментарий (он обязателен)
- Жмем `Commit and Push`


trainer
master
Commit
Changes
Unversioned Files 6 files
.gitignore .idea
.gitignore
App.java src/main/java/ru/akhitev
AppTest.java src/test/java/ru/akhitev
pom.xml
vcs.xml .idea
1
Amend
3 added
init maven
2
3
Commit
Commit and Push...
Git
Log
Console

GIT

КОММИТИМ НАШИ ИЗМЕНЕНИЯ

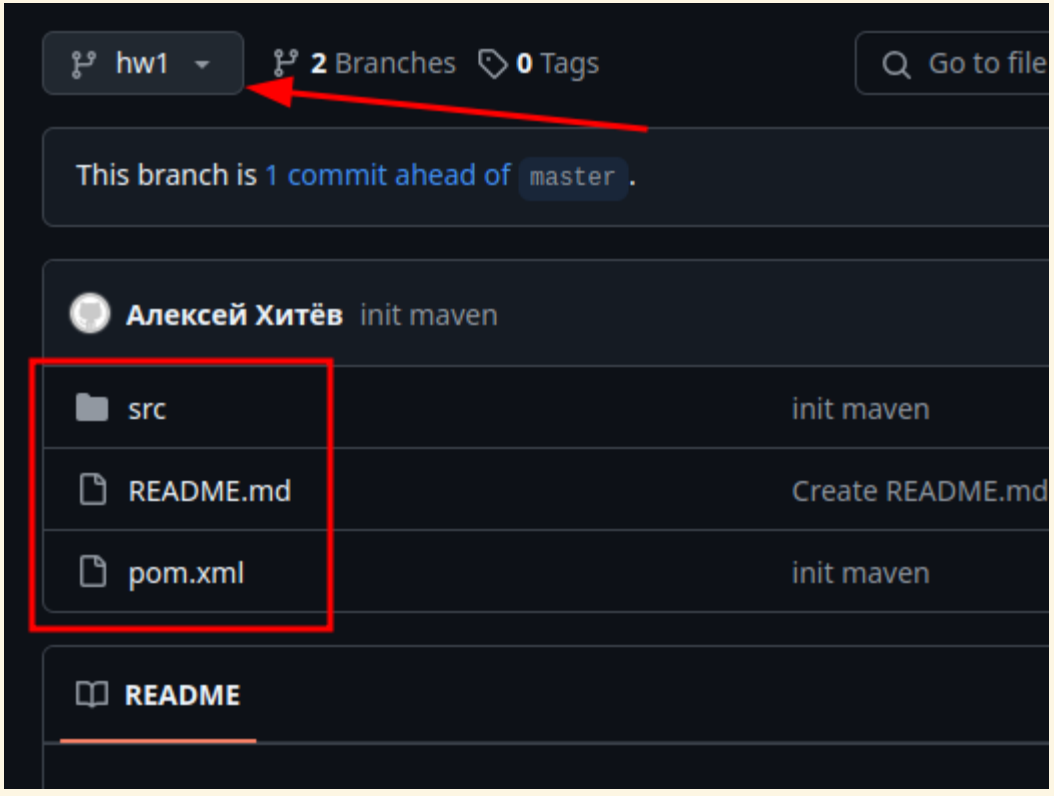
- Проверяем, что не закоммитили ничего лишнего
- Жмем `Push`


Push Commits to trainer
hw1 → origin : hw1
init maven
trainer 3 files
src 2 files
main/java/ru/akhitev 1 file
App.java
test/java/ru/akhitev 1 file
AppTest.java
pom.xml
For Commit and Push to non-protected branches, preview commits before push
Push tags:
All
Push
Cancel

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`

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" x
  <modelversion>4.0.0</modelversion>

  <groupid>ru.akhitev</groupid>
  <artifactid>trainer</artifactid>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>

  ...
</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

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" x
  <modelversion>4.0.0</modelversion>

  <groupid>ru.akhitev</groupid>
  <artifactid>trainer</artifactid>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>

  ...
</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

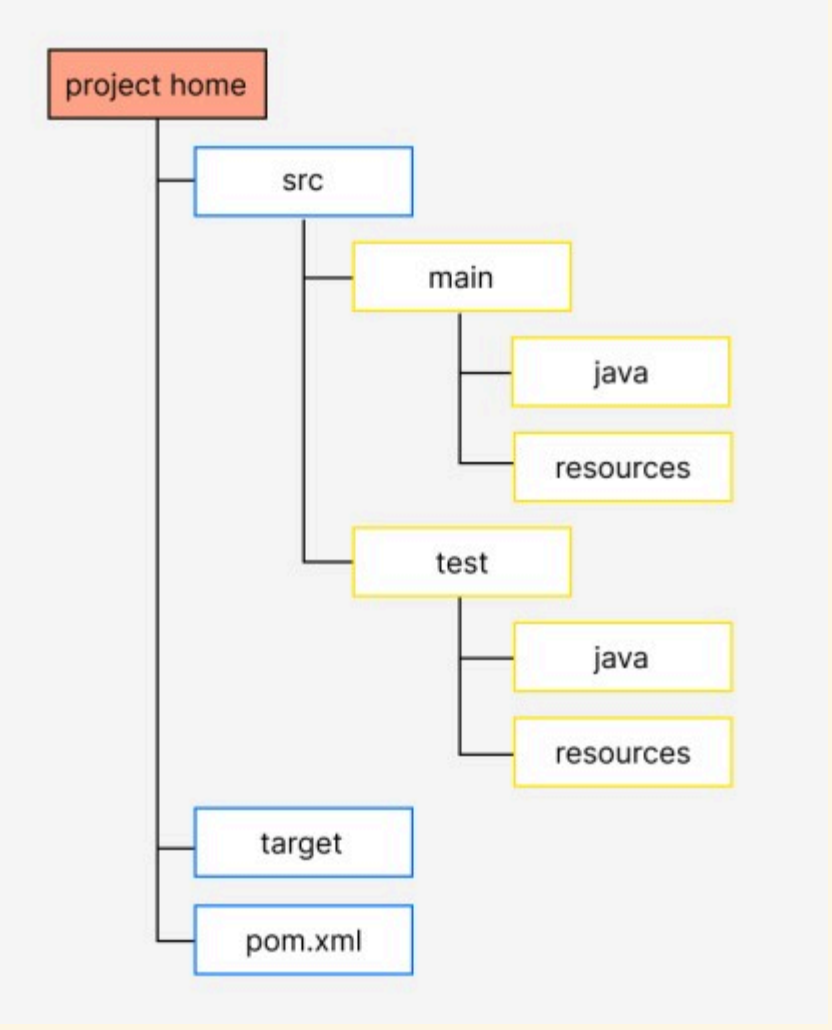
ЗАВИСИМОСТИ

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" x
  ...

  <dependencies>
    <dependency>
      <groupid>junit</groupid>
      <artifactid>junit</artifactid>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</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



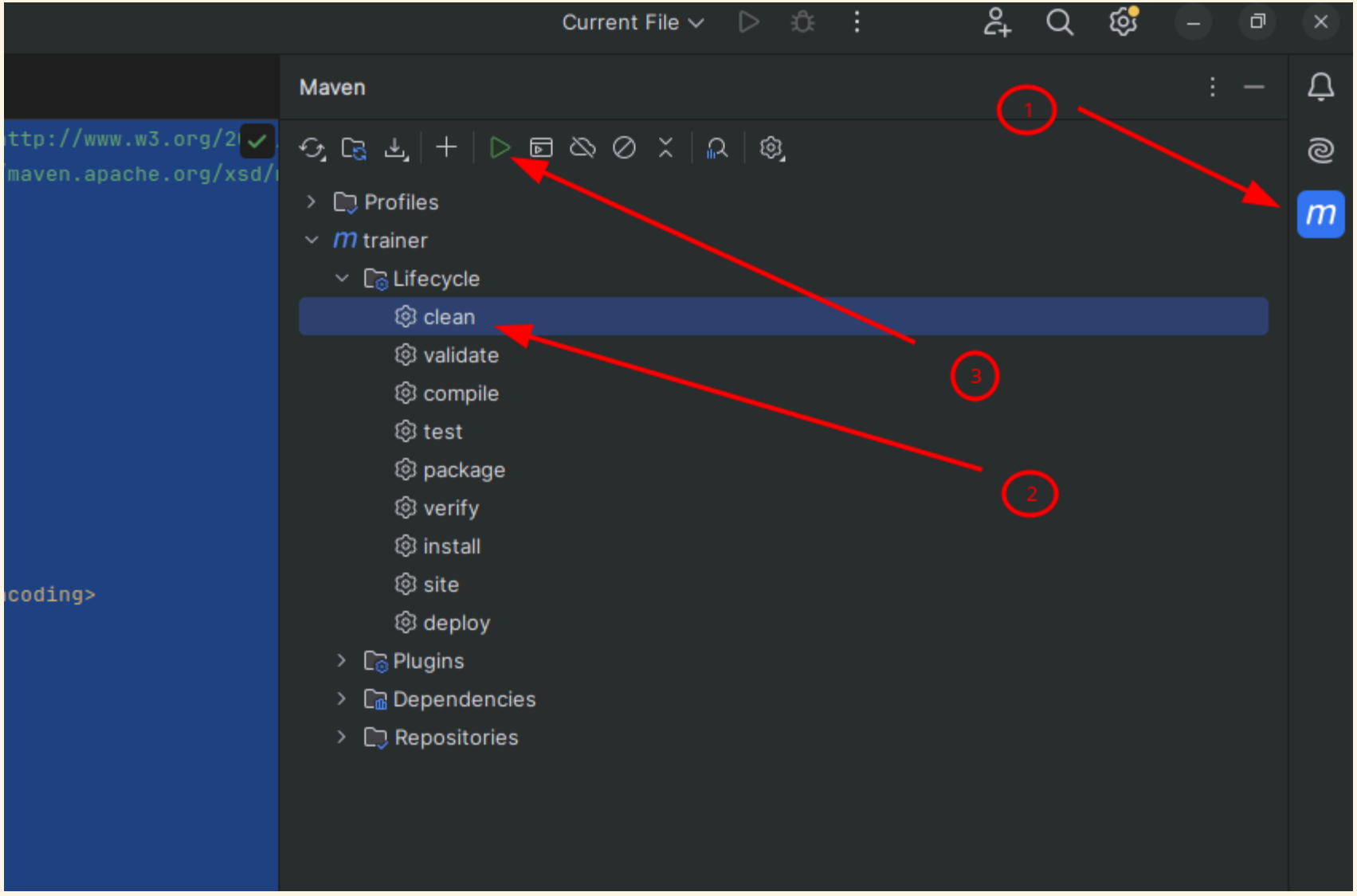
КОМАНДЫ MAVEN

CLEAN

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

```
mvn clean
```

- Если в Idea, то


Current File
Maven
Profiles
trainer
Lifecycle
clean
validate
compile
test
package
verify
install
site
deploy
Plugins
Dependencies
Repositories
1
2
3

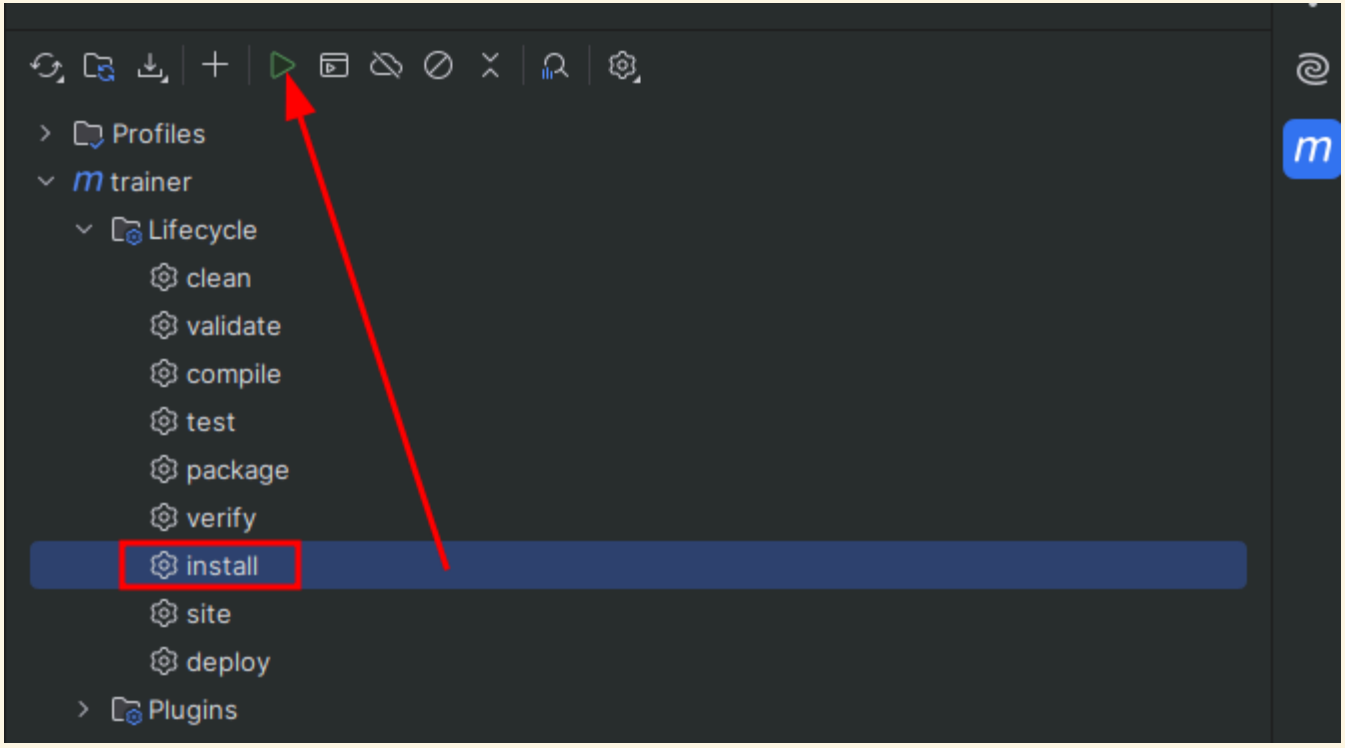
КОМАНДЫ MAVEN

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


Profiles
trainer
Lifecycle
clean
validate
compile
test
package
verify
install
site
deploy
Plugins

КОМАНДЫ MAVEN

ПРОЧИЕ

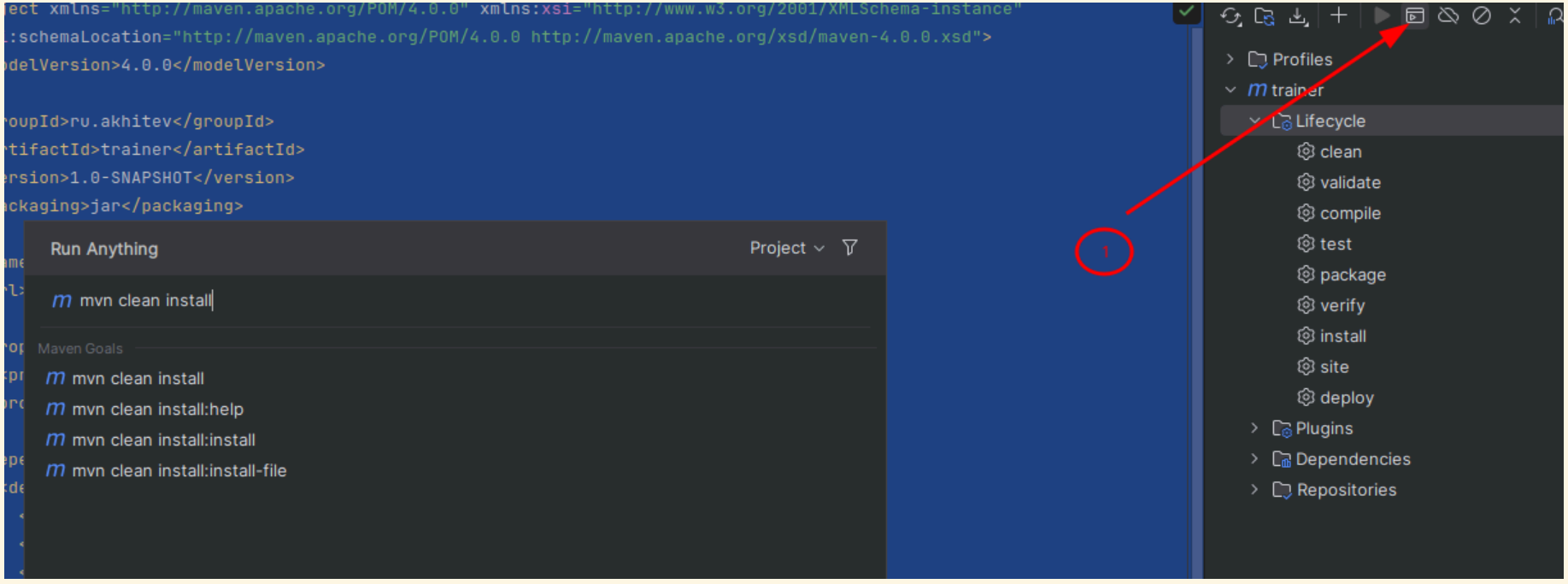
- `compile` - компилирует исходный код
- `package` только собирает проект и создает артефакт
- `validate` - проверяет проект на правильность и что всего хватает для сборки
- и другие

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

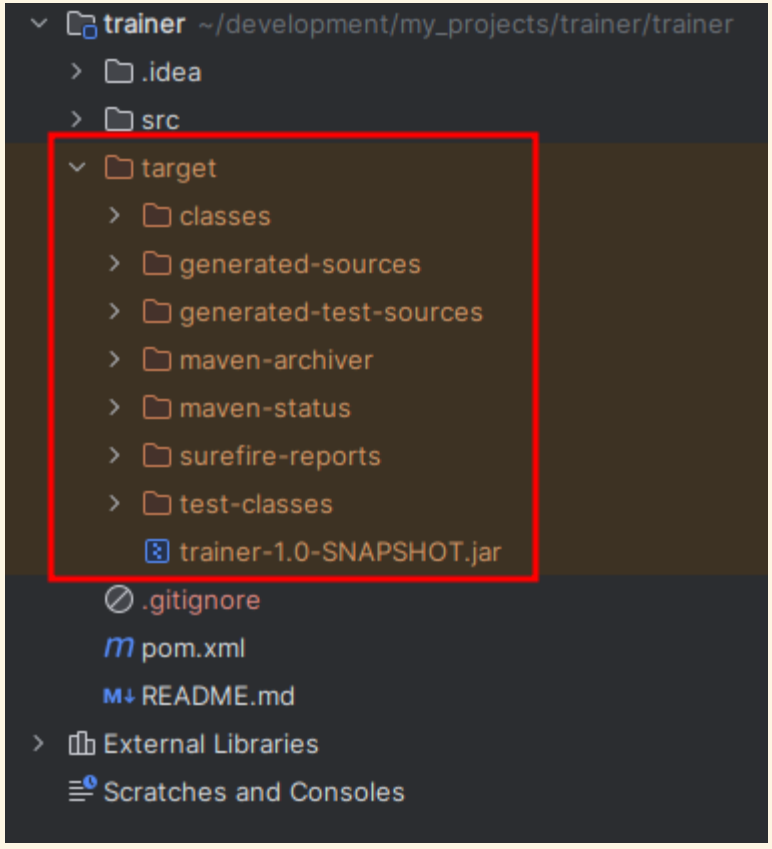
ГДЕ ЛОКАЛЬНЫЙ РЕПОЗИТОРИЙ?

- 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 видимо, `groupdId`, `artefact` и `version` преобразовались в папки в пути к репозиторию
- В папке `.m2` еще лежит файлик `settings.xml`. Там описываются данные для доступа к удаленным репозиториям

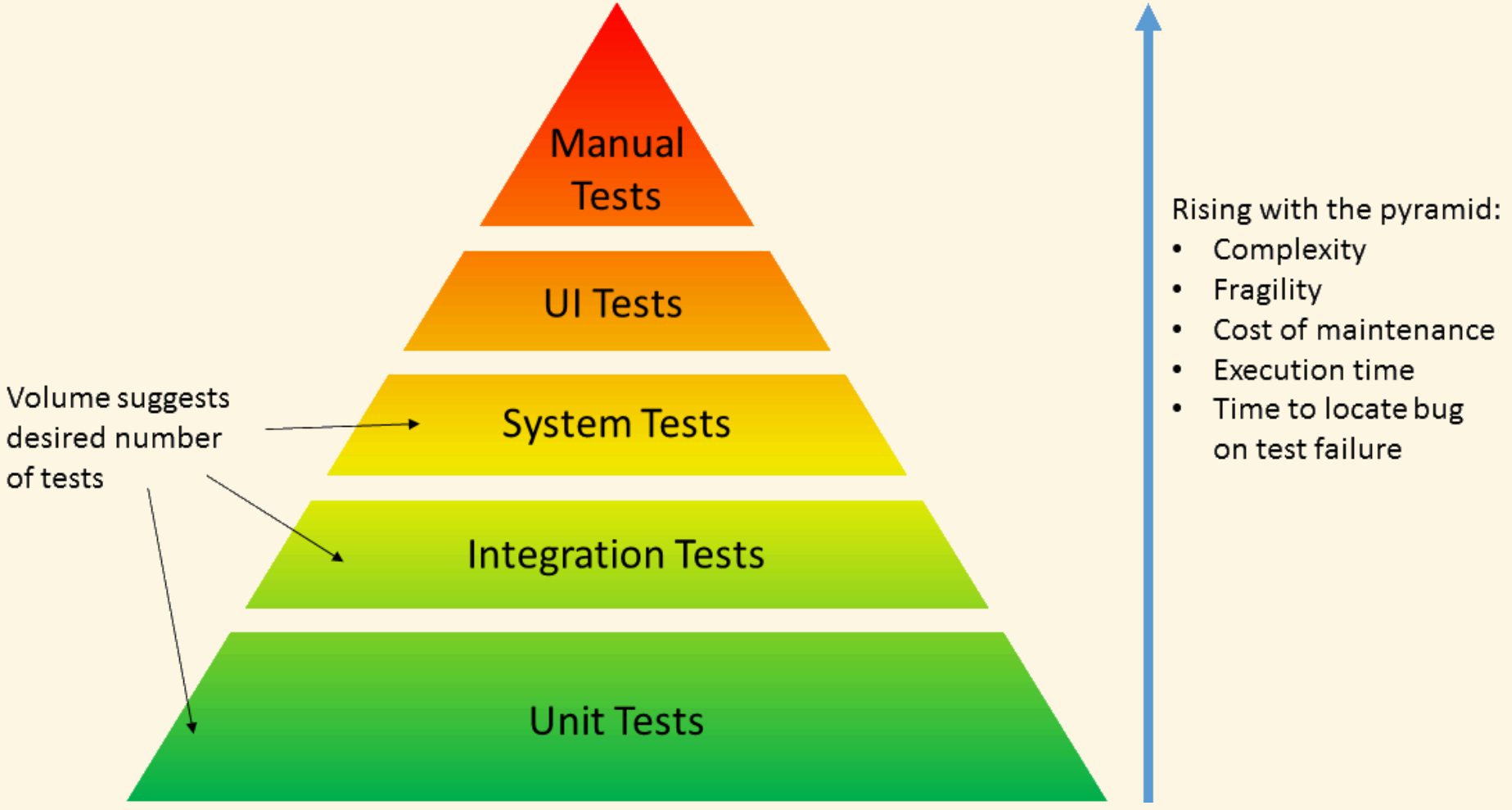
УДАЛЕННЫЕ РЕПОЗИТОРИИ

- 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, убирам 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 автотестировщиками
- 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

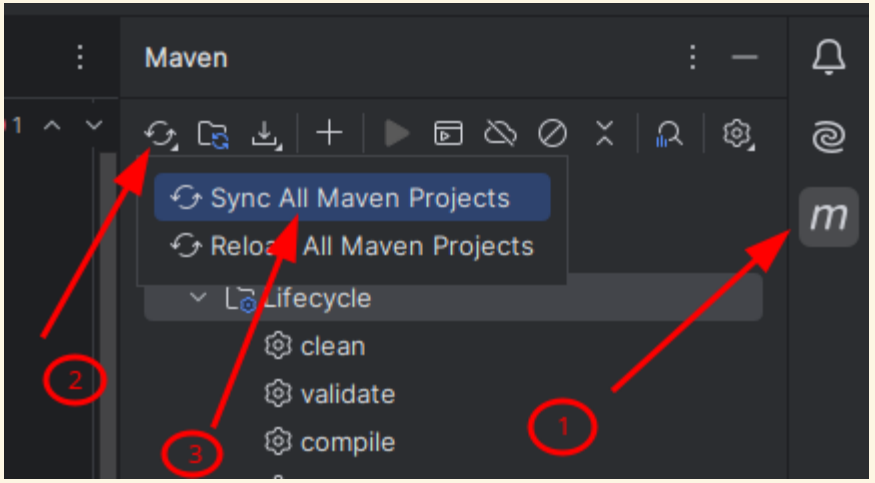
JUNIT5

- Обновленный JUnit, содержащий необходимые инструкменты 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`

```xml
<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

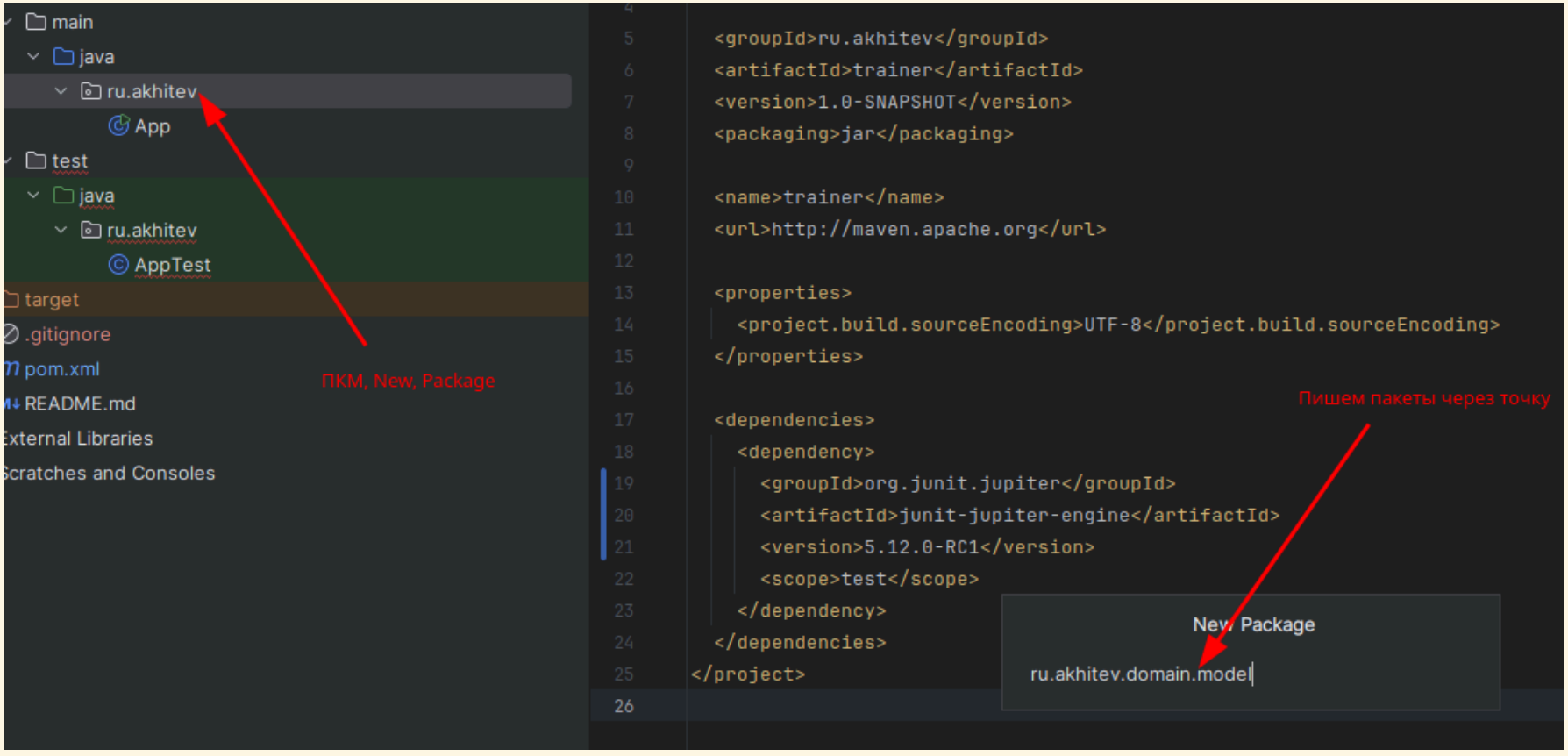
- Чтобы зависимость “проросла” в проекте, нужно выполнить синхронизацию проекта


Maven
Sync All Maven Projects
Reload All Maven Projects
Lifecycle
clean
validate
compile
1
2
3

СДЕЛАЕМ ТЕСТИРУЕМЫЙ КЛАСС

ДОБАЛЯЕМ ПАКЕТ

- Добавим пакет `domain` и 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`

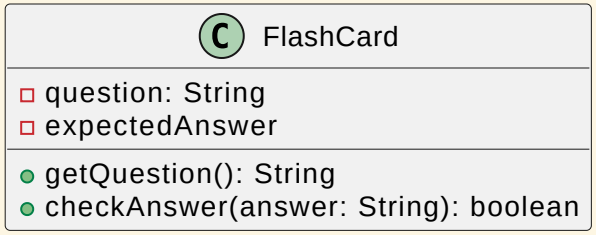


- В новом пакете создаем класс `FlashCard`

СДЕЛАЕМ ТЕСТИРУЕМЫЙ КЛАСС

ДОБАЛЯЕМ САМ КЛАСС

- В пакете `ru.<ваш_ник>.domain.model` создаем класс `FlashCard`
  Для этого нужно нажать правой кнопкой на пакете `ru.<ваш_ник>` и выбрать `New`, `Class`

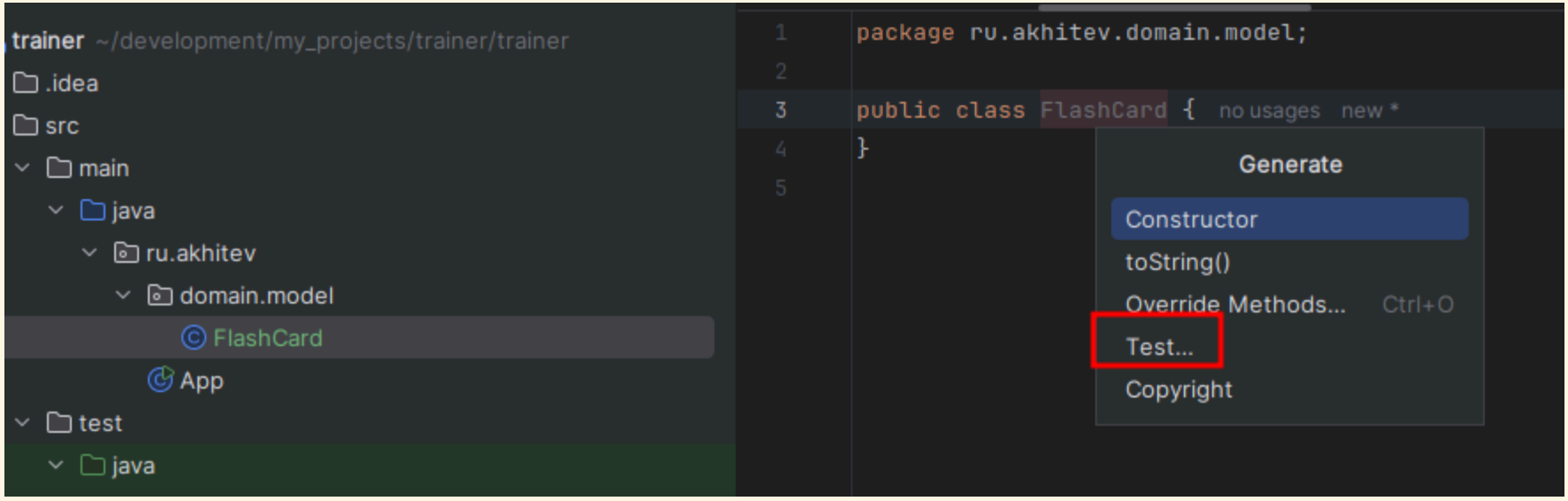


- По тексту примерно так

```
public class FlashCard {
  private final String question;
  private final String expectedAnswer;

  public FlashCard(String question, String expectedAnswer) {
    this.question = question;
    this.expectedAnswer = expectedAnswer;
  }

  public String getQuestion() {
    return question;
  }

  public boolean checkAnswer(String answer) {
    return answer.equals(expectedAnswer);
```

ДЕЛАЕМ КЛАСС ДЛЯ UNIT-ТЕСТОВ

- Идем в `src/test` и удаляем там сгенерированный класс `AppTest`
- Открываем наш класс `FlashCard` 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`

НАПИШЕМ ПЕРВЫЙ ТЕСТ

HAPPY PATH

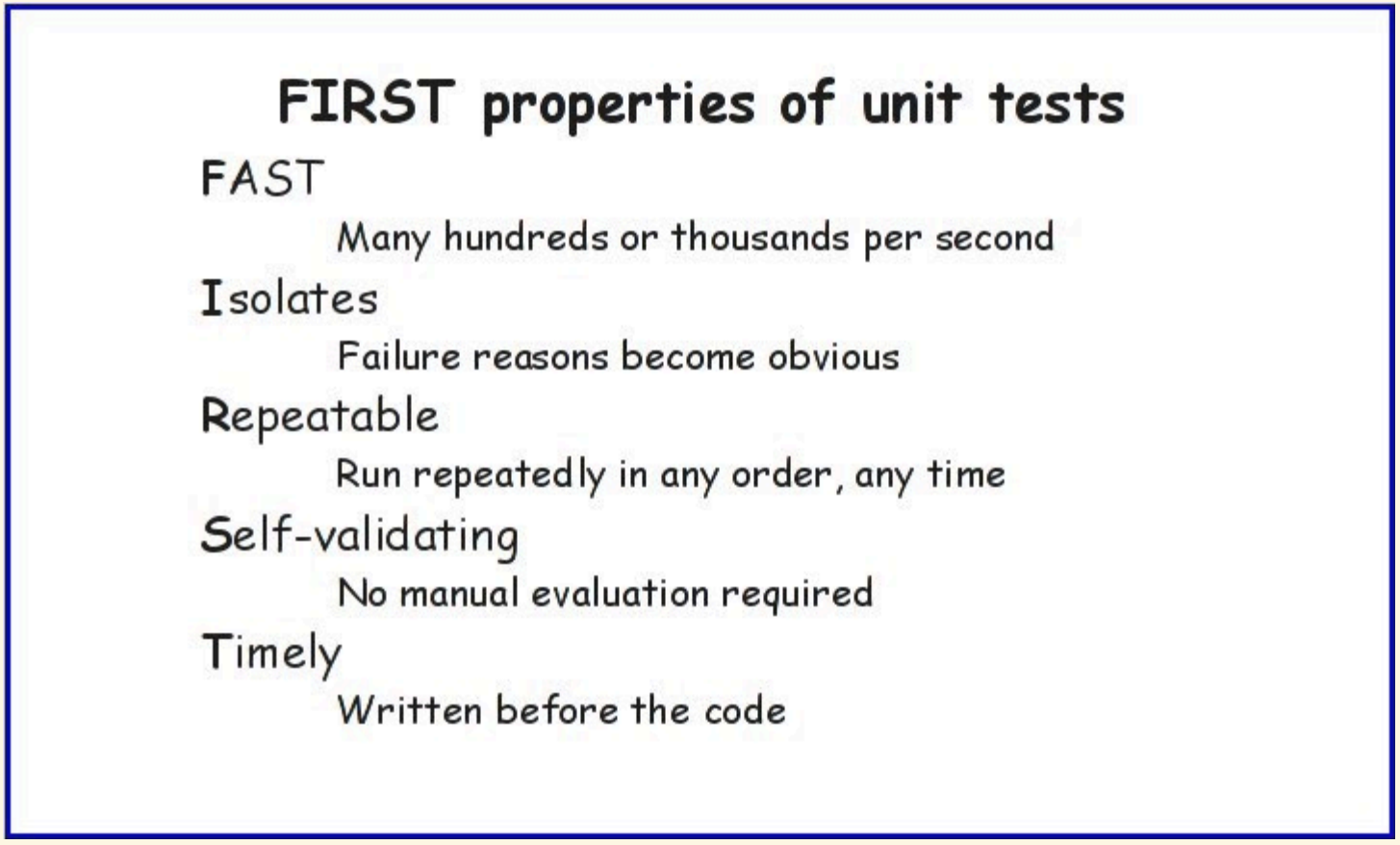
```java
package ru.akhitev.domain.model;

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

import static org.junit.jupiter.api.Assertions.*;
class FlashCardTest {

  @Test
  @DisplayName("checkAnswer возвращает true при правильном ответе")
  void when_AnswerIsOk_then_checkAnswer_isTrue() {
    String question = "Вежливо попросить что то сделать на английском";
    String expectedAnswer = "Would You like to do something?";
    FlashCard card = new FlashCard(question, expectedAnswer);
    assertTrue(card.checkAnswer("Would You like to do something?"));
  }
}
```

НАПИШЕМ ЕЩЕ ТЕСТЫ

НАЧНЕМ ВСЕ ЛОМАТЬ!

- На каждый кейс пишем отдельный unit-тест!
- Если ответ будет непрваильным, 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

ПРИНЦИП F.I.R.S.T



ЗАКОММИТИМ ИЗМЕНЕНИЯ

- Выбираем файлы в коммит
- Пишем комментарий
- Жмем `Commit And Push`
- Проверяем GitHub

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.