

# Maven и Gradle

Урок 1









## Евгений Манько

Java-разработчик, создатель данного курса

- ✨ Разрабатывал бэкенд для Яндекс, Тинькофф, МТС;
- ✨ Победитель грантового конкурса от “Росмолодежь”
- ✨ Руководил IT-Департаментом “Студенты Москвы”
- ✨ И т.д.



## Что будет на уроке сегодня

-  Основные принципы систем сборки и их роли в разработке программного обеспечения
-  Краткий обзор и сравнение Maven и Gradle
-  Теория и практика Maven
-  Теория и практика Gradle
-  Сравнение Maven и Gradle
-  Заключение



## Введение. Системы сборки

Maven и Gradle — две популярные системы сборки.



**Maven** использует XML для структурирования проектов и предоставляет широкий набор плагинов. Он следует принципу "соглашение против конфигурации".



**Gradle**, использующий Groovy или Kotlin DSL, является более гибким и настраиваемым инструментом, особенно популярным в разработке Android-приложений.



Gradle

VS



Maven



## Введение. Системы сборки

В курсе будут рассмотрены основы обеих систем сборки, их возможности и функции, такие как:

- управление зависимостями,
- тестирование,
- генерация документации,
- пакетирование и распространение,
- поддержка плагинов.

# Краткий обзор и сравнение Maven и Gradle

Maven и Gradle — две мощные системы сборки, каждая со своими особенностями и преимуществами. Давайте разберем их вместе.

Критерий	Maven	Gradle
Структура проекта	XML-структура (pom.xml)	Groovy или Kotlin DSL (build.gradle или build.gradle.kts)
Читабельность	Менее читабельный из-за XML	Более читабельный благодаря DSL
Скорость	Стабильный, но медленнее	Быстрый и производительный
Гибкость	Меньше гибкости, ограничения	Больше гибкости и возможностей



# Основные понятия Maven

## **POM (Project Object Model)**

это XML-файл, который описывает проект и его настройки.

## **Зависимости**

это внешние библиотеки или модули, необходимые для корректной работы вашего приложения.

## **Плагины**

это компоненты, которые расширяют функциональность Maven, добавляя дополнительные задачи и интеграцию с другими инструментами.

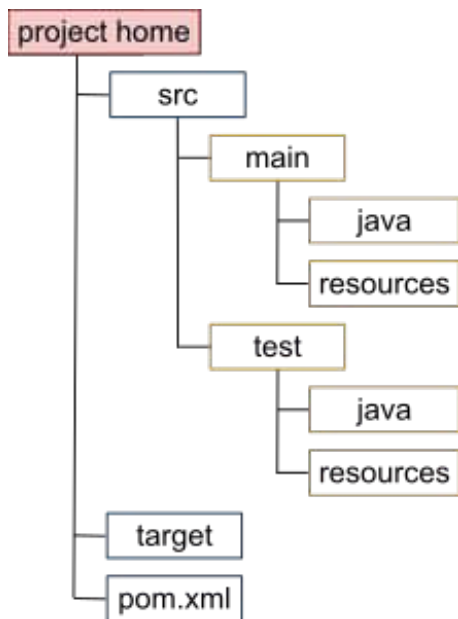
## **Жизненный цикл**

это последовательность фаз, определяющих процесс сборки проекта.



# Структура проекта Maven

Стандартная структура проекта Maven состоит из следующих каталогов и файлов:



- 💡 `src/main/java`: Здесь хранится исходный код вашего приложения.
- 💡 `src/main/resources`: Здесь находятся ресурсы вашего приложения, такие как файлы конфигурации, изображения и т. д.
- 💡 `src/test/java`: Здесь размещается исходный код тестов вашего приложения.
- 💡 `src/test/resources`: Здесь хранятся ресурсы, связанные с тестированием.
- 💡 `pom.xml`: Это основной файл конфигурации Maven, содержащий информацию о проекте и его настройках.





# Жизненные циклы Maven

1

## Clean

Этот жизненный цикл отвечает за удаление всех файлов, созданных в результате предыдущей сборки.

2

## Default

Этот жизненный цикл является основным и отвечает за сборку, тестирование, пакетирование и развертывание вашего проекта.

3

## Site

Этот жизненный цикл отвечает за создание документации и сайта вашего проекта. Он включает фазы pre-site, site и post-site.



## Основные фазы жизненного цикла default

- ✓ **Validate:** В этой фазе проверяется корректность настроек проекта и отсутствие проблем с конфигурацией.
- ✓ **Compile:** В этой фазе исходный код проекта компилируется в байт-код Java.
- ✓ **Test:** В этой фазе выполняются тесты вашего приложения. Обратите внимание, что этот этап не вызывает остановку сборки в случае неудачных тестов.
- ✓ **Package:** В этой фазе создается артефакт вашего проекта (например, JAR-файл).
- ✓ **Verify:** В этой фазе выполняются проверки, чтобы убедиться, что пакет соответствует качеству и критериям проекта.
- ✓ **Install:** В этой фазе артефакт вашего проекта устанавливается в локальный репозиторий Maven, чтобы быть доступным для других проектов на вашем компьютере.
- ✓ **Deploy:** В этой фазе артефакт вашего проекта развертывается в удаленном репозитории, чтобы быть доступным для других разработчиков и команд.



## Зависимости Maven

Зависимости — это внешние библиотеки или модули, которые требуются для корректной работы вашего приложения. В Maven зависимости объявляются в POM-файле, и система автоматически управляет ими, загружая необходимые артефакты из репозиторий и интегрируя их в ваш проект.

```
1 <dependencies>
2   <dependency>
3     <groupId>com.example</groupId>
4     <artifactId>my-library</artifactId>
5     <version>1.0.0</version>
6   </dependency>
7 </dependencies>
```



## Репозитории Maven

Репозитории — это централизованные хранилища артефактов, таких как библиотеки и плагины. В Maven существует три типа репозиторий: локальный, центральный, удаленный.

```
1 <repositories>
2     <repository>
3         <id>example-repo</id>
4         <url>https://example.com/repo</url>
5     </repository>
6 </repositories>
```



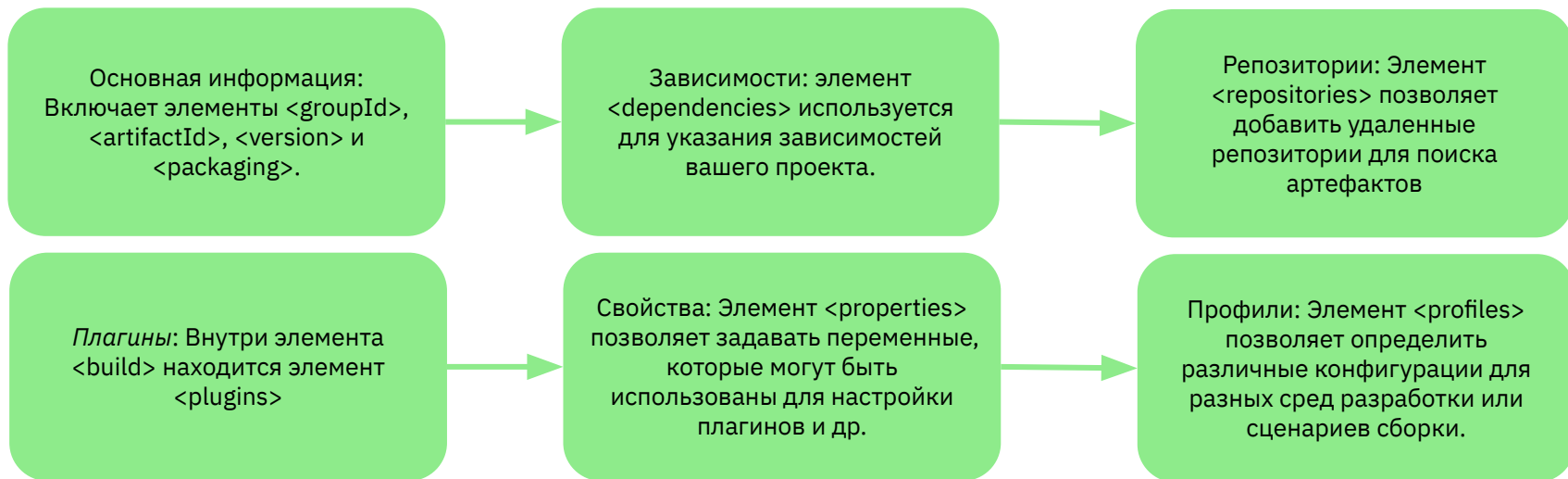
## Плагины Maven

Плагины — это расширения Maven, предоставляющие дополнительные функции и возможности для управления процессом сборки вашего проекта. Плагины состоят из одной или нескольких задач (goals), которые могут быть вызваны в различных фазах жизненного цикла сборки.

```
1 <build>
2   <plugins>
3     <plugin>
4       <groupId>org.apache.maven.plugins</groupId>
5       <artifactId>maven-compiler-plugin</artifactId>
6       <version>3.8.1</version>
7       <configuration>
8         <!-- Настройка плагина -->
9       </configuration>
10    </plugin>
11  </plugins>
12 </build>
```

# Настройка проекта Maven

POM-файл является основой для настройки проекта в Maven. В POM-файле вы можете указать информацию о проекте, такую как группа, идентификатор артефакта, версия, а также настроить зависимости, плагины, репозитории и другие элементы.





## Создание проекта с помощью Maven

1

Сначала установите Maven, следуя инструкциям на официальном сайте: <https://maven.apache.org/install.html>.

2

Для создания простого Java-проекта выполните следующую команду:

```
mvn archetype:generate  
-DgroupId=com.mycompany.app  
-DartifactId=my-app  
-DarchetypeArtifactId=maven-archetype-quickstart  
-DinteractiveMode=false
```

3

Здесь `com.mycompany.app` — это пример `groupId`, а `my-app` — `artifactId` вашего проекта.



## Структура проекта

В созданной директории my-app вы увидите следующую структуру проекта:

```
1 |— pom.xml
2 |-- src
3 |   |— main
4 |       |-- java
5 |           |-- com
6 |               |-- mycompany
7 |                   |-- app
8 |                       |-- App.java
9 |   |-- test
10 |       |-- java
11 |           |-- com
12 |               |-- mycompany
13 |                   |-- app
14 |                       |-- AppTest.java
```





## Сборка и запуск проекта

1

Чтобы собрать ваш проект, перейдите в директорию my-app и выполните следующую команду:

```
mvn package
```

2

Для запуска собранного приложения выполните команду:

```
java -cp  
target/my-app-1.0-SNAPSHOT.jar  
com.mycompany.app.App
```

3

Эта команда запустит класс App из пакета com.mycompany.app. Если все сделано правильно, вы увидите следующий вывод:  
Hello, World!



## Добавление зависимостей и плагинов

Откройте pom.xml и добавьте следующий код внутри элемента <dependencies>

```
1 <dependency>
2   <groupId>org.apache.commons</groupId>
3   <artifactId>commons-lang3</artifactId>
4   <version>3.12.0</version>
5 </dependency>
```



## Добавление зависимостей и плагинов

Добавьте следующий код внутри элемента `<plugins>`

```
1 <plugin>
2   <groupId>org.apache.maven.plugins</groupId>
3   <artifactId>maven-assembly-plugin</artifactId>
4   <version>3.3.0</version>
5   <configuration>
6     <archive>
7       <manifest>
8         <mainClass>com.mycompany.app.App</mainClass>
9       </manifest>
10    </archive>
11    <descriptorRefs>
12      <descriptorRef>jar-with-dependencies</descriptorRef>
13    </descriptorRefs>
14  </configuration>
15  <executions>
16    <execution>
17      <id>make-assembly</id>
18      <phase>package</phase>
19      <goals>
20        <goal>single</goal>
21      </goals>
22    </execution>
23  </executions>
24 </plugin>
```

## Добавление зависимостей и плагинов

Maven создаст исполняемый JAR-файл с именем `my-app-1.0-SNAPSHOT-jar-with-dependencies.jar`, который содержит все зависимости вашего проекта.

Для запуска этого JAR-файла выполните команду:

```
java -jar target/my-app-1.0-SNAPSHOT-jar-with-dependencies.jar
```

Теперь у вас есть полноценный Java-проект, созданный и собранный с помощью Maven.



## Добавление зависимостей

Для добавления зависимостей в ваш Maven-проект вам нужно отредактировать файл `pom.xml`. Зависимости добавляются внутри тега `<dependencies>`. В качестве примера добавим библиотеку Google Guava:

```
1 <dependency>
2   <groupId>com.google.guava</groupId>
3   <artifactId>guava</artifactId>
4   <version>30.1-jre</version>
5 </dependency>
```

Здесь `groupId`, `artifactId` и `version` определяют уникальный идентификатор и версию библиотеки. Теперь вы можете использовать классы и методы из библиотеки Google Guava в своем проекте.



## Настройка плагинов

Настроим плагин `maven-compiler-plugin` для использования Java 11:

```
1 <build>
2   <plugins>
3     <plugin>
4       <groupId>org.apache.maven.plugins</groupId>
5       <artifactId>maven-compiler-plugin</artifactId>
6       <version>3.8.1</version>
7       <configuration>
8         <source>11</source>
9         <target>11</target>
10      </configuration>
11    </plugin>
12  </plugins>
13 </build>
```



## Работа с профилями

Для создания профиля, добавьте следующий код внутри тега `<profiles>` в вашем `pom.xml` файле:

Чтобы активировать профиль, используйте ключ `-P` при выполнении команды `mvn`, например:  
`mvn package -P development`

Эта команда активирует профиль `development` и применит его настройки во время сборки проекта.

```
1 <profiles>
2   <profile>
3     <id>development</id>
4     <build>
5       <plugins>
6         <!-- Здесь могут быть плагины, используемые
           только в режиме разработки -->
7       </plugins>
8     </build>
9   </profile>
10  <profile>
11    <id>production</id>
12    <build>
13      <plugins>
14        <!-- Здесь могут быть плагины, используемые
           только в режиме продакшн -->
15      </plugins>
16    </build>
17  </profile>
18 </profiles>
```



## Настройка ресурсов

Добавьте следующий код внутри тега `<plugins>`

```
1 <plugin>
2   <groupId>org.apache.maven.plugins</groupId>
3   <artifactId>maven-resources-plugin</artifactId>
4   <version>3.2.0</version>
5   <configuration>
6     <encoding>UTF-8</encoding>
7     <resources>
8       <resource>
9         <directory>src/main/resources</directory>
10        <includes>
11          <include>**/*.properties</include>
12          <include>**/*.xml</include>
13        </includes>
14      </resource>
15    </resources>
16  </configuration>
17 </plugin>
```





## Добавление зависимости для работы с базами данных

Для добавления Hibernate в ваш проект, вставьте следующий код внутри тега `<dependencies>` файла `pom.xml`

```
1 <dependency>
2   <groupId>org.hibernate</groupId>
3   <artifactId>hibernate-core</artifactId>
4   <version>5.4.32.Final</version>
5 </dependency>
```



## Настройка плагина для анализа кода

В качестве примера настроим плагин Checkstyle.

Для этого добавьте следующий код внутри тега `<plugins>`:

```
1 <plugin>
2   <groupId>org.apache.maven.plugins</groupId>
3   <artifactId>maven-checkstyle-plugin</artifactId>
4   <version>3.1.2</version>
5   <executions>
6     <execution>
7       <id>validate</id>
8       <phase>validate</phase>
9       <goals>
10        <goal>check</goal>
11      </goals>
12    </execution>
13  </executions>
14  <configuration>
15    <configLocation>checkstyle.xml</configLocation>
16    <encoding>UTF-8</encoding>
17    <consoleOutput>true</consoleOutput>
18    <failsOnError>true</failsOnError>
19  </configuration>
20 </plugin>
```



## Добавление плагина для тестирования

Сначала добавьте зависимости:

```
1 <dependency>
2   <groupId>org.junit.jupiter</groupId>
3   <artifactId>junit-jupiter-api</artifactId>
4   <version>5.8.1</version>
5   <scope>test</scope>
6 </dependency>
7 <dependency>
8   <groupId>org.junit.jupiter</groupId>
9   <artifactId>junit-jupiter-engine</artifactId>
10  <version>5.8.1</version>
11  <scope>test</scope>
12 </dependency>
```



## Добавление плагина для тестирования

Затем добавьте плагин `maven-surefire-plugin` для запуска тестов JUnit внутри тега `<plugins>`:

```
1 <plugin>
2   <groupId>org.apache.maven.plugins</groupId>
3   <artifactId>maven-surefire-plugin</artifactId>
4   <version>3.0.0-M5</version>
5   <configuration>
6     <includes>
7       <include>**/*Test.java</include>
8     </includes>
9   </configuration>
10 </plugin>
```



## Добавление плагина для создания исполняемого JAR-файла

Для этого вам потребуется настроить плагин `maven-shade-plugin`. Добавьте следующий код внутри тега `<plugins>`:

```
1 <plugin>
2   <groupId>org.apache.maven.plugins</groupId>
3   <artifactId>maven-shade-plugin</artifactId>
4   <version>3.2.4</version>
5   <executions>
6     <execution>
7       <phase>package</phase>
8       <goals>
9         <goal>shade</goal>
10      </goals>
11      <configuration>
12        <transformers>
13          <transformer
14            implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
15            <mainClass>com.example.MainClass</mainClass>
16          </transformer>
17        </transformers>
18      </configuration>
19    </execution>
20  </executions>
21 </plugin>
```



## Создание собственного плагина

**1.** Сначала создайте новый Maven-проект для вашего плагина.  
В файле `pom.xml` указать `maven-plugin` в качестве `packaging`-типа:

```
<packaging>maven-plugin</packaging>
```

**2.** Чтобы создать плагин, вам понадобится Maven Plugin API.  
Добавьте следующую зависимость в тег `<dependencies>`:

```
1 <dependency>
2   <groupId>org.apache.maven</groupId>
3   <artifactId>maven-plugin-api</artifactId>
4   <version>3.8.4</version>
5 </dependency>
```



## Создание собственного плагина

Вам также могут понадобиться следующие зависимости для компиляции и тестирования плагина:

```
1 <dependency>
2   <groupId>org.apache.maven.plugin-tools</groupId>
3   <artifactId>maven-plugin-annotations</artifactId>
4   <version>3.6.2</version>
5   <scope>provided</scope>
6 </dependency>
7 <dependency>
8   <groupId>org.apache.maven</groupId>
9   <artifactId>maven-core</artifactId>
10  <version>3.8.4</version>
11  <scope>provided</scope>
12 </dependency>
```



# Создание собственного плагина

## 3. Создание класса плагина.

Создайте новый Java-класс, который будет представлять ваш плагин. Этот класс должен расширять `org.apache.maven.plugin.AbstractMojo`. Например, создайте класс `MyPluginMojo`







## Создание собственного плагина

### 4. Сборка плагина

Теперь соберите плагин с помощью команды `mvn package`. Если все настроено правильно, вы получите JAR-файл с вашим плагином в директории `target`.

### 5. Использование плагина в проекте

Чтобы использовать ваш плагин в другом Maven-проекте, добавьте его в раздел `<plugins>` файла `pom.xml`:

```
1 <plugin>
2   <groupId>com.example</groupId>
3   <artifactId>my-maven-plugin</artifactId>
4   <version>1.0.0</version>
5   <executions>
6     <execution>
7       <id>run-my-plugin</id>
8       <phase>validate</phase>
9       <goals>
10        <goal>my-plugin</goal>
11      </goals>
12    </execution>
13  </executions>
14 </plugin>
```



## Структура проекта Gradle

Структура проекта Gradle похожа на структуру Maven-проекта, но есть некоторые различия.  
Вот типичная структура Gradle-проекта:

```
1 |— build.gradle
2 |— settings.gradle
3 |— gradle/
4 |   |— wrapper/
5 |       |— gradle-wrapper.jar
6 |       |— gradle-wrapper.properties
7 |— gradlew
8 |— gradlew.bat
9 |— src/
10 |   |— main/
11 |       |— java/
12 |       |— resources/
13 |   |— test/
14 |       |— java/
15 |       |— resources/
```



# Структура проекта Gradle

1

**build.gradle:** Главный файл сборки, где вы определяете все настройки, плагины, зависимости и задачи для вашего проекта. Этот файл написан на Groovy или Kotlin DSL (Domain-Specific Language).

2

**settings.gradle:** Файл настроек, который содержит информацию о многомодульных проектах и дополнительные настройки. Здесь вы можете указать имя проекта и включить подпроекты.

3

**gradle/wrapper:** Директория, содержащая Gradle Wrapper, инструмент, который позволяет пользователям собирать проект, не устанавливая Gradle локально.

4

**gradlew и gradlew.bat:** Исполняемые файлы оболочки Gradle Wrapper для Unix-подобных и Windows-систем соответственно. Эти файлы позволяют разработчикам запускать Gradle-сборку без предварительной установки Gradle на своем компьютере.



## Стандартные задачи Gradle



**compileTestJava:** Компилирует исходный код модульных тестов.



**processTestResources:** Копирует ресурсы, используемые во время модульного тестирования, в директорию сборки.



**testClasses:** Задача-агрегатор, которая зависит от `compileTestJava` и `processTestResources`. Выполняется после завершения обеих предыдущих задач.



**test:** Запускает модульные тесты и генерирует отчеты о результатах тестирования. Обычно выполняется после завершения задачи `testClasses`.



**check:** Задача-агрегатор, которая зависит от `test` и других задач, связанных с проверкой качества кода (например, статическим анализом кода или проверкой стиля). Выполняется после завершения задачи `test`.



**build:** Задача-агрегатор, которая зависит от `assemble` и `check`. Выполняется после завершения всех связанных задач и является конечной задачей, связанной со сборкой проекта.



## Зависимости Gradle

Для управления зависимостями в Gradle используется файл сценария сборки `build.gradle`. Зависимости определяются внутри блока `dependencies`. Вот пример добавления зависимости на библиотеку “Guava”:

```
dependencies {  
    implementation 'com.google.guava:guava:30.1-jre'  
}
```





## Конфигурации Gradle



### **implementation:**

Зависимости,  
необходимые  
для компиляции и  
выполнения  
приложения.



### **compileOnly:**

Зависимости,  
необходимые только  
для компиляции  
приложения, но не  
включаемые в сборку.



### **runtimeOnly:**

Зависимости,  
необходимые только  
во время выполнения  
приложения.



### **testImplementation:**

Зависимости,  
необходимые  
для компиляции  
и выполнения  
модульных тестов.



## Репозитории Gradle

Репозитории — это удаленные хранилища, в которых хранятся артефакты зависимостей. Gradle предоставляет поддержку для различных типов репозиториев, таких как Maven, Ivy и даже файловые репозитории.

Для добавления репозитория в проект, вам нужно определить его в блоке `repositories` файла сценария сборки `build.gradle`. Вот пример добавления репозитория Maven Central:

```
repositories {  
    mavenCentral()  
}
```



## Плагины Gradle

Плагины в Gradle — это расширения, которые добавляют новые функции и задачи в ваш проект. Вам может потребоваться плагин для работы с определенным типом проекта, интеграции с внешними системами или автоматизации специфичных для вашего проекта задач.

Плагины могут быть применены в файле сценария сборки `build.gradle` с помощью метода `apply()`. Вот пример применения плагина Java:

```
apply plugin: 'java'
```

Также вы можете использовать новый синтаксис `plugins` для применения плагинов:

```
plugins {  
    id 'java'  
}
```







## Настройка проекта Gradle

Настройка проекта обычно выполняется в файле сценария сборки build.gradle.

```
1 apply plugin: 'java'
2
3 group = 'com.example'
4 version = '1.0.0'
5
6 repositories {
7     mavenCentral()
8 }
9
10 dependencies {
11     implementation 'com.google.guava:guava:30.1-jre'
12 }
13
14 tasks.withType(JavaCompile) {
15     options.encoding = 'UTF-8'
16 }
```



# Создание простого Java проекта с помощью Gradle

1

## Инициализация проекта

Для начала создайте новую папку для вашего проекта и перейдите в нее с помощью терминала. Затем выполните следующую команду для инициализации Gradle-проекта с использованием шаблона Java-проекта:

```
gradle init --type java-application
```

Эта команда создаст структуру каталогов проекта, файл `build.gradle` со стандартной конфигурацией и пример Java-класса с методом `main()`.

2

## Структура проекта

Структура каталогов проекта должна выглядеть примерно так:

```
my-java-project/  
├─ src/  
│   ├── main/  
│   │   └─ java/  
│   │       └─ App.java  
│   └─ test/  
│       └─ java/  
│           └─ AppTest.java  
├─ build.gradle  
└─ settings.gradle
```



## Создание простого Java проекта с помощью Gradle

3

### Сборка проекта

Чтобы собрать проект, выполните следующую команду в корневом каталоге проекта:

```
./gradlew build
```

Эта команда выполнит компиляцию исходного кода, сборку артефактов и модульное тестирование. Результат сборки будет помещен в каталог build.

4

### Запуск приложения

Чтобы запустить ваше приложение, выполните следующую команду:

```
./gradlew run
```

Это выполнит задачу run, которая запускает Java-приложение с использованием собранных классов и зависимостей. Вы должны увидеть вывод “Hello, World!” в терминале.

# Добавление зависимостей Gradle

## 1. Указание репозитория:

В файле build.gradle добавьте блок repositories, в котором указываются репозитории для поиска зависимостей.

Например, чтобы добавить Maven Central, добавьте следующий код:

```
repositories {  
    mavenCentral()  
}
```

## 2. Объявление зависимостей:

В файле build.gradle добавьте блок dependencies, в котором указываются зависимости вашего проекта. Например, чтобы добавить зависимость на библиотеку “Guava”, добавьте следующий код:

```
dependencies {  
    implementation  
    'com.google.guava:guava:30.1-jre'  
}
```



# Настройка плагинов Gradle

## 1. Применить плагин:

В файле `build.gradle` добавьте код для применения плагина. Например, чтобы применить плагин “Java”, добавьте следующий код:

```
apply plugin: 'java'
```

или используйте синтаксис `plugins`:

```
plugins {  
    id 'java'  
}
```





# Настройка плагинов Gradle

## 2. Конфигурировать плагин:

После применения плагина, вы можете настроить его поведение с помощью специальных блоков и методов. Например, чтобы указать версию Java и включить сжатие JAR-файла, добавьте следующий код:

```
1 java {
2     sourceCompatibility = '1.8'
3     targetCompatibility = '1.8'
4 }
5
6 jar {
7     manifest {
8         attributes 'Implementation-Title': 'My Java Project', 'Implementation-Version': version
9     }
10    zip64 = true
11 }
```

# Использование профилей для разных сборок

## 1. Создание переменной окружения

Для начала, создайте переменную окружения, которая будет указывать на текущий профиль сборки.

Вам нужно добавить эту переменную в файле `.gradle/gradle.properties`:

```
profile=default
```

Здесь `default` — это имя профиля по умолчанию. Вы можете заменить его на другое имя, если хотите использовать другой профиль.





# Использование профилей для разных сборок

## 2. Создание блока профилей

В файле build.gradle создайте блок профилей с разными настройками для каждого профиля.

```
1 ext.profiles = [  
2     default: {  
3         applicationName = 'MyAppDefault'  
4     },  
5     production: {  
6         applicationName = 'MyAppProduction'  
7     },  
8     development: {  
9         applicationName = 'MyAppDevelopment'  
10    }  
11 ]
```



# Использование профилей для разных сборок

## 3. Применение профиля

Чтобы применить текущий профиль, используйте следующий код в файле build.gradle:

```
ext.profileConfig = profiles[project.findProperty('profile') ?: 'default']
```

Этот код выбирает профиль на основе значения переменной окружения profile. Если переменная не задана, используется профиль по умолчанию.



## 4. Использование значений профиля

Теперь вы можете использовать значения из текущего профиля в вашем сценарии сборки.

```
jar {  
    manifest {  
        attributes 'Implementation-Title': profileConfig.applicationName, 'Implementation-Version': version  
    }  
}
```

# Использование профилей для разных сборок



## 5. Сборка с разными профилями

Для сборки вашего проекта с разными профилями, передайте значение переменной окружения `profile` через командную строку:

```
./gradlew -Pprofile=production build
```

Эта команда выполнит сборку с профилем `production`.

Это упрощает управление конфигурациями вашего проекта и позволяет легко адаптировать его для различных сред разработки, тестирования и развертывания. В дальнейшем, вы сможете применять эти знания для создания еще более сложных и мощных сценариев сборки с использованием Gradle.

# Создание собственного плагина на Java

## 1. Создание нового Java-проекта

Для начала создайте новый Java-проект для разработки плагина. Вы можете использовать любой инструмент, который вам нравится, такой как IntelliJ IDEA или Eclipse. Не забудьте инициализировать проект с помощью Gradle.

## 2. Добавление зависимостей

Чтобы создать плагин Gradle, вам нужно добавить зависимость на Gradle API в файле build.gradle:

```
dependencies {  
    implementation 'org.gradle:gradle-api:7.3.3'  
}
```

Замените 7.3.3 на актуальную версию Gradle, если это необходимо.

## Создание собственного плагина на Java

### 3. Создание класса плагина

Создайте новый Java-класс для вашего плагина, который реализует интерфейс `org.gradle.api.Plugin`.

```
1 import org.gradle.api.Plugin;
2 import org.gradle.api.Project;
3
4 public class MyCustomPlugin implements Plugin<Project> {
5     @Override
6     public void apply(Project project) {
7         // Код плагина здесь
8     }
9 }
```

# Создание собственного плагина на Java

## 4. Реализация логики плагина

В методе `apply()` добавьте логику вашего плагина. Например, вы можете создать новую задачу:

```
1 import org.gradle.api.DefaultTask;
2 import org.gradle.api.Plugin;
3 import org.gradle.api.Project;
4 import org.gradle.api.tasks.TaskAction;
5
6 public class MyCustomPlugin implements Plugin<Project> {
7     @Override
8     public void apply(Project project) {
9         project.getTasks().create("myCustomTask", MyCustomTask.class);
10    }
11 }
12
13 class MyCustomTask extends DefaultTask {
14     @TaskAction
15     public void myCustomAction() {
16         System.out.println("Hello from MyCustomTask!");
17     }
18 }
```

# Создание собственного плагина на Java

## 5. Сборка и публикация плагина

Чтобы собрать ваш плагин, выполните команду `./gradlew build`. Результатом сборки будет JAR-файл, который можно использовать в других проектах.

Если вы хотите опубликовать ваш плагин в репозитории (например, в Maven Central или JCenter), вам нужно добавить необходимые настройки и плагины в ваш файл `build.gradle`. Для подробных инструкций обратитесь к документации Gradle.

## 6. Использование собственного плагина

Чтобы использовать ваш плагин в другом проекте, добавьте его JAR-файл в каталог `libs` этого проекта и добавьте зависимость в файл `build.gradle`:

```
1 buildscript {
2     repositories {
3         flatDir {
4             dirs 'libs'
5         }
6     }
7     dependencies {
8         classpath files('libs/MyCustomPlugin.jar')
9     }
10 }
11
12 apply plugin: 'my.custom.plugin'
```

# Сравнение Maven и Gradle

## MAVEN

- + Функциональность
- + Интеграция с существующими инструментами и платформами
- + Обучение и сообщество
- + Внедрение

## GRADLE

- + Производительность
- + Скорость сборки
- + Конфигурация и настройка
- + Функциональность
- + Гибкость
- + Интеграция с существующими инструментами и платформами
- + Внедрение
- + Обучение и использование

## Возможные проблемы

1. Сложность настройки

2. Проблемы с зависимостями

3. Производительность сборки

4. Неправильное использование плагинов

5. Изменения в экосистеме

6. Командная работа



## Рекомендации по выбору между Maven и Gradle

1. Опыт и знания команды

4. Интеграция с другими инструментами

2. Требования проекта

5. Поддержка и сообщество

3. Производительность сборки



**Спасибо за внимание**

