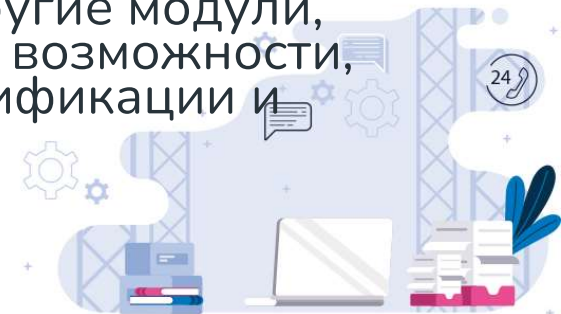


# Итоги лекции 1

- Платформа Spring — это широко используемая платформа Java с открытым исходным кодом, которая предоставляет комплексную модель программирования и конфигурации для создания корпоративных приложений. Его архитектура построена на основе двух основных принципов: внедрения зависимостей (DI) и аспектно-ориентированного программирования (AOP).
- Платформа Spring состоит из нескольких модулей, которые можно разделить на четыре основные области: основной контейнер, доступ к данным/интеграция, Интернет и прочее. Базовый контейнер обеспечивает фундаментальную функциональность платформы Spring, включая контейнер IoC и ApplicationContext. Область доступа к данным/интеграции обеспечивает поддержку интеграции с базами данных и другими источниками данных. Веб-область обеспечивает поддержку создания веб-приложений, включая модули Spring MVC и Spring WebFlux. Область «Разное» включает в себя другие модули, предоставляющие дополнительные функциональные возможности, например модуль Spring Security для функций аутентификации и авторизации.



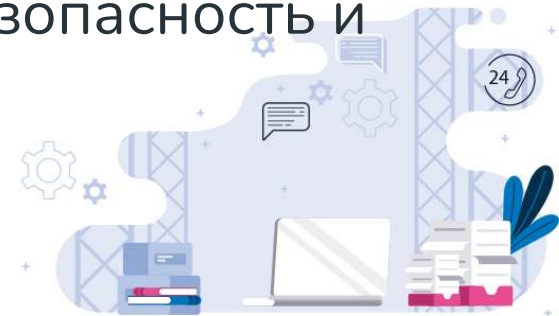
# Особенности

- Внедрение зависимостей (DI) — это шаблон проектирования, который помогает уменьшить связь между компонентами приложения. Используя DI, среда Spring обеспечивает слабую связь между компонентами, что делает приложение более модульным и простым в обслуживании. Платформа Spring предоставляет контейнер инверсии управления (IoC), который отвечает за создание экземпляров JavaBeans и управление ими, а также ApplicationContext, который обеспечивает унифицированное представление всей конфигурации приложения.



# Особенности

- Аспектно-ориентированное программирование (АОП) позволяет разработчикам модульно объединять сквозные задачи, такие как ведение журнала, безопасность и управление транзакциями, и применять их к нескольким компонентам приложения. Это приводит к созданию более модульной и многократно используемой базы кода. Платформа Spring предоставляет структуру АОП, которая отвечает за управление сквозными проблемами приложения, такими как ведение журнала, безопасность и управление транзакциями.



# Особенности

- В целом архитектура среды Spring основана на принципах модульности, разделения задач и гибкости, предоставляя разработчикам мощный набор инструментов для создания надежных, масштабируемых и поддерживаемых корпоративных приложений. Модульная архитектура платформы позволяет разработчикам выбирать только необходимые модули для своих конкретных нужд, сокращая ненужные накладные расходы и сложность приложения. Кроме того, гибкая модель конфигурации платформы Spring позволяет разработчикам настраивать приложение, используя различные подходы, такие как
  - ❖ конфигурация на основе XML,
  - ❖ конфигурация на основе Java
  - ❖ конфигурация на основе аннотаций.



# Виды конфигураций Spring-приложений

- Groovy-based (для фанатов).
- XML-based (классика, но устарела).
- Annotation-based
  - ❑ Java-based (@Bean)
  - ❑ Annotation-based (@Autowired, @Service, @Controller)



# 10 причин использовать Spring Framework

1. Легкий, простой и легкий
2. Создает безопасные веб-приложения.
3. Шаблон MVC
4. Простота связи с базами данных
5. Модульная конструкция
6. Может быть интегрирован с другими платформами.
7. Внедрение зависимостей
8. Следует аспектно-ориентированному программированию
9. Тестирование становится простым
10. Легко обращайтесь с внешними ресурсами



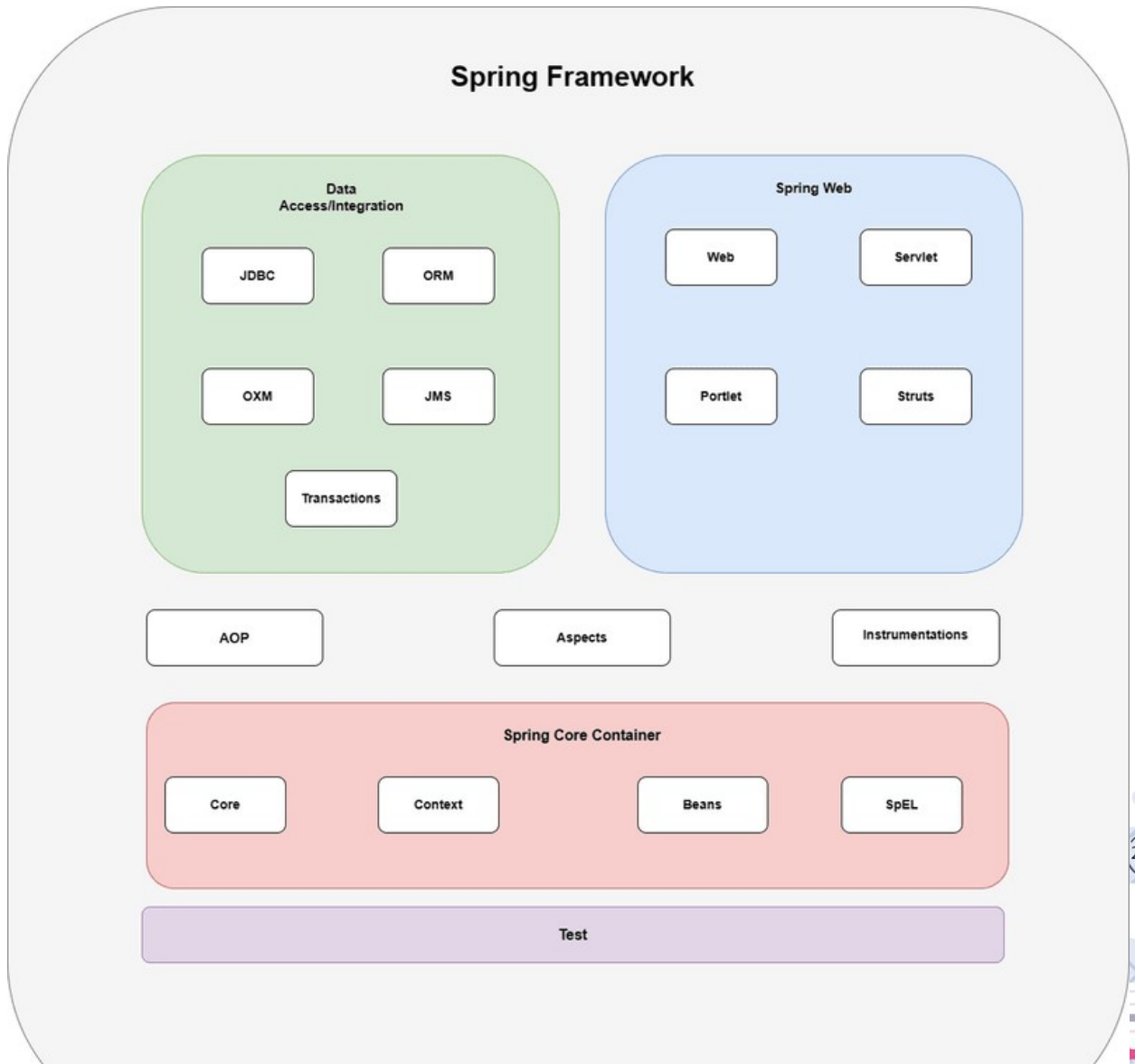
# 10 самых распространенных ошибок Spring Framework

1. Неправильная настройка
2. Неэффективное использование внедрения зависимостей
3. Неправильная обработка исключений
4. Конфликты версий
5. Разделение ответственности
6. Автосвязывание иногда может не работать, если оно используется неправильно, например, если создание нового экземпляра выполняется вручную вместо использования Spring или класс, для которого мы используем автосвязывание
7. Написание повторяющегося кода
8. Проблема с конфигурацией на основе XML
9. Больше внимания уделять задачам низкого уровня
10. Проблема с портом



# Spring Modules

Платформа Spring является модульной и состоит из нескольких модулей, которые предоставляют различные функциональные возможности, помогающие создавать корпоративные приложения. Модули можно условно разделить на четыре основные области: **основной контейнер, доступ к данным/интеграция, Интернет и прочее.**



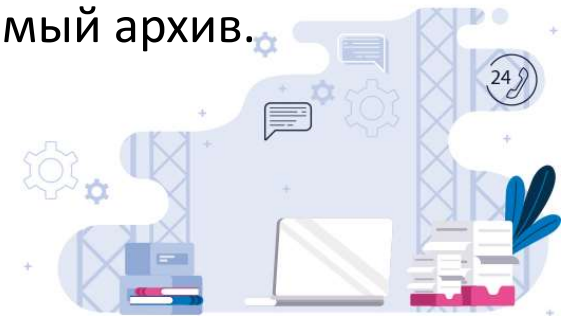


# Система сборки

Система сборки — это программное обеспечение, которое упрощает сборку приложений. Систему сборки можно настроить на автоматическое выполнение различных задач, являющихся частью процесса сборки, вместо того чтобы разбираться с ними вручную.

Вот несколько примеров задач, которые часто выполняются при сборке приложения:

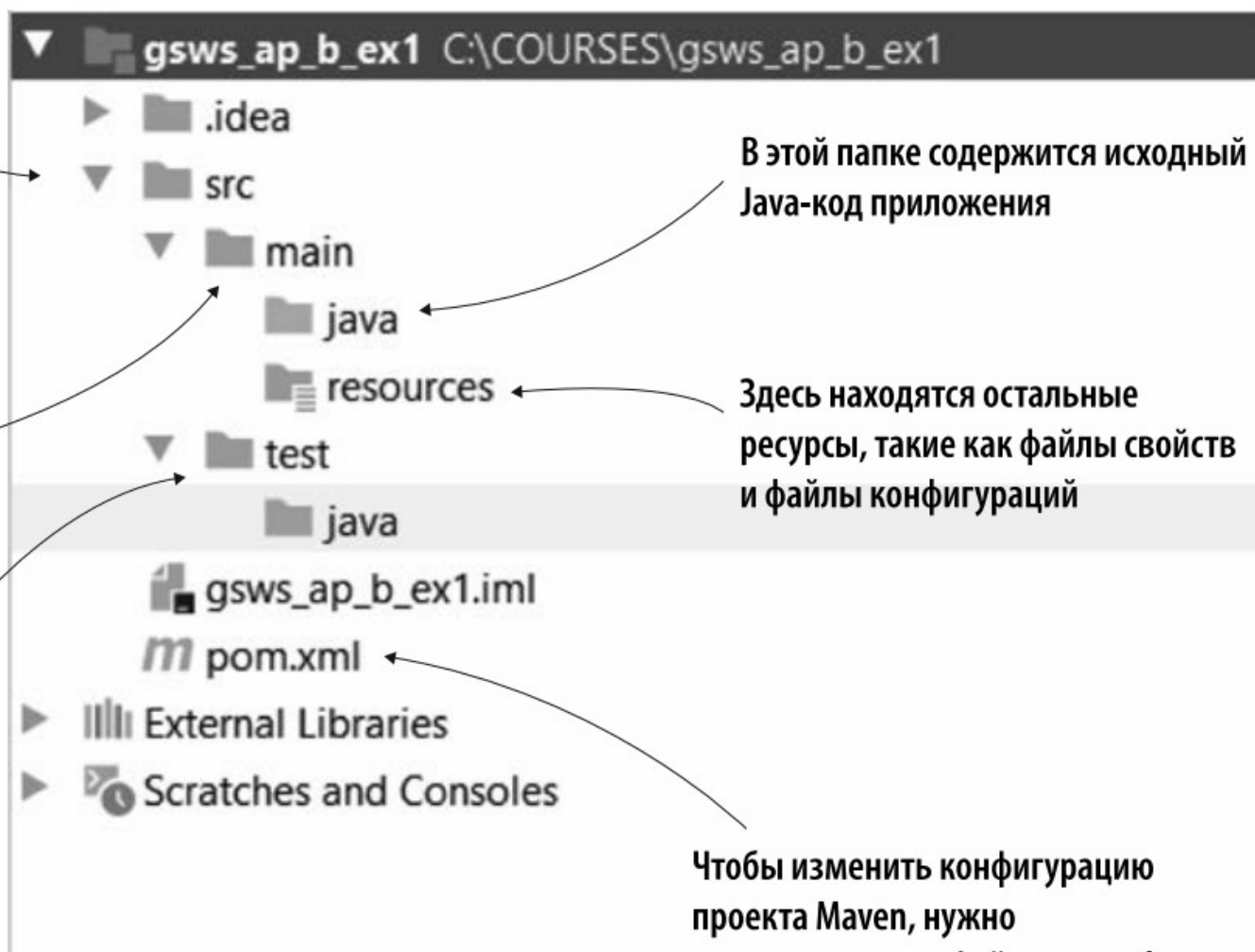
- ❖ загрузка зависимостей, необходимых для работы приложения;
- ❖ выполнение тестов;
- ❖ проверка соответствия синтаксиса заданным правилам;
- ❖ проверка на наличие уязвимостей в системе безопасности;
- ❖ компиляция приложения; упаковка приложения в исполняемый архив.



В этой папке хранится  
весь исходный код

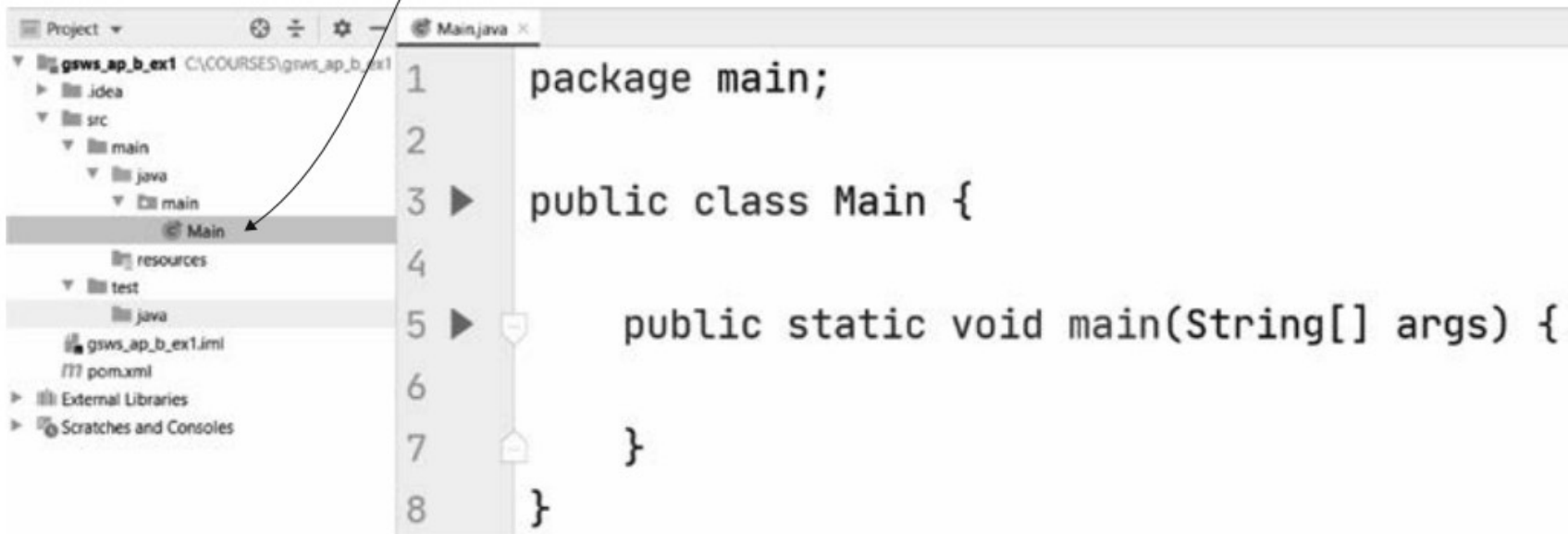
В этой папке лежит  
исходный код  
приложения

Здесь хранится  
исходный код  
модульных тестов



Чтобы изменить конфигурацию  
проекта Maven, нужно  
отредактировать файл pom.xml

В папке Java находятся обычные пакеты и классы Java, которые создаются для проекта. В данном случае я создал пакет main и новый класс Main внутри него



# Содержимое файла pom.xml по умолчанию

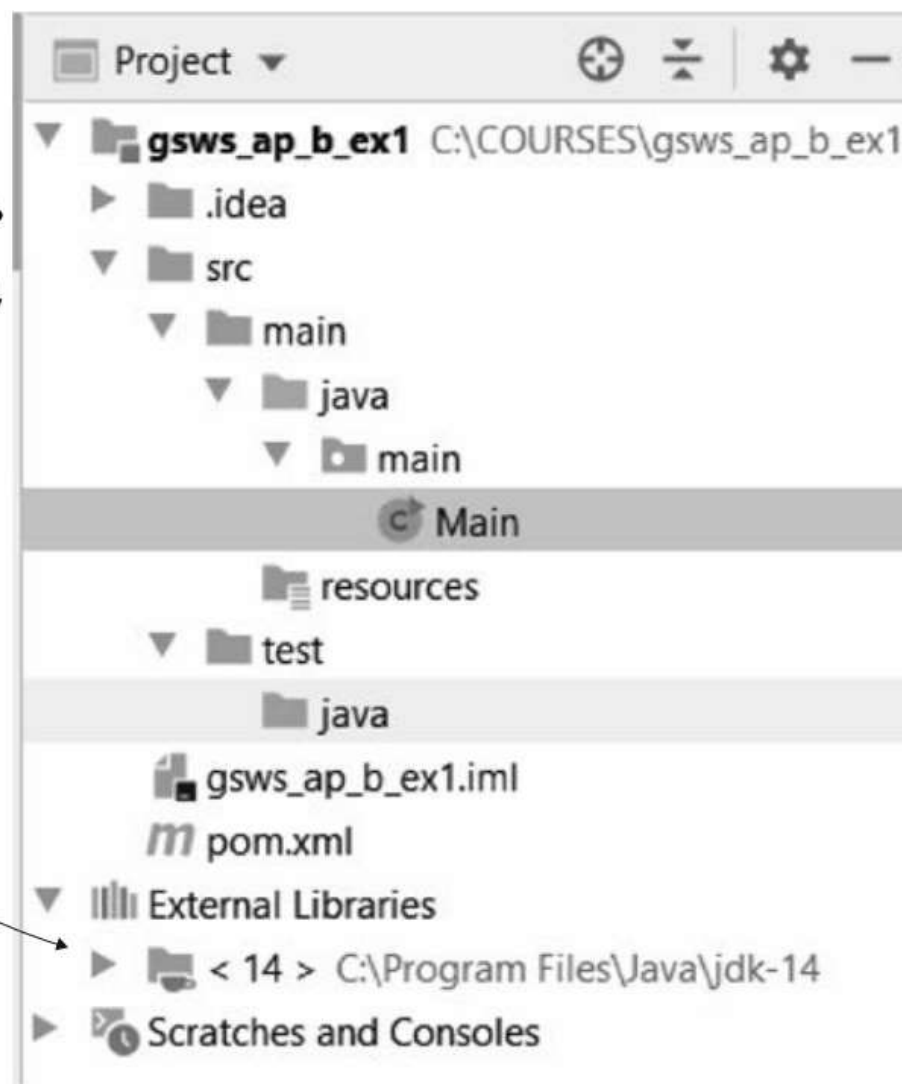
```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://Maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://Maven.apache.org/POM/4.0.0
    http://Maven.apache.org/xsd/Maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>

  <groupId>org.example</groupId>
  <artifactId>sq-ch2-ex1</artifactId>
  <version>1.0-SNAPSHOT</version>
</project>
```



Пока в разделе External Libraries нового проекта есть только JDK. По мере добавления в проект зависимостей здесь будут появляться новые файлы, соответствующие внешним зависимостям



# Добавление зависимости в файл pom.xml

```
<?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>org.example</groupId>
  <artifactId>sq_ch2_ex1</artifactId>
  <version>1.0-SNAPSHOT</version>
```

Все зависимости проекта  
записываются между тегами  
`<dependencies>` и `</dependencies>`

```
  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-jdbc</artifactId>
```

Описание зависимости  
представляет собой группу тегов  
`<dependency> ... </dependency>`

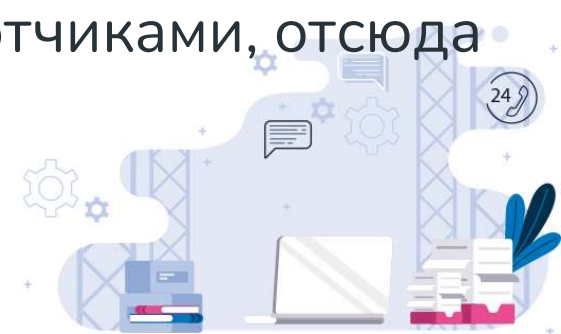
```
      <version>5.2.6.RELEASE</version>
    </dependency>
  </dependencies>
</project>
```





# Inversion of Control(IoC) – Инверсия управления

Контейнер Spring IoC (инверсия управления) — это ядро [Spring Framework](#) . Он создает объекты, настраивает и собирает их зависимости, управляет всем их жизненным циклом. Контейнер использует внедрение зависимостей (DI) для управления компонентами, составляющими приложение. Он получает информацию об объектах из файла конфигурации (XML), кода Java или аннотаций Java и класса Java POJO. Эти объекты называются Beans. Поскольку управление объектами Java и их жизненным циклом не осуществляется разработчиками, отсюда и название «Инверсия управления».



# Существует 2 типа контейнеров IoC:

- BeanFactory
- Контекст приложения

Это означает, что если вы хотите использовать контейнер IoC ?  
нужно ли нам использовать BeanFactory или  
ApplicationContext. BeanFactory — это самая базовая версия  
контейнеров IoC, а ApplicationContext расширяет возможности  
BeanFactory.

Ниже приведены некоторые основные особенности Spring IoC.

- Создание объекта для нас,
- Управление нашими объектами,
- Помогая нашему приложению быть настраиваемым,
- Управление зависимостями





# Контекст приложения

- Контейнер Spring IoC отвечает за создание экземпляров, подключение, настройку и управление всем жизненным циклом объектов. BeanFactory и ApplicationContext представляют контейнеры Spring IoC. ApplicationContext — это дополнительный интерфейс BeanFactory. BeanFactory предоставляет базовые функции и рекомендуется использовать для легких приложений, таких как мобильные устройства и апплеты. ApplicationContext предоставляет базовые функции в дополнение к следующим корпоративным функциям:
- Публикация событий зарегистрированным прослушивателям путем разрешения файлов свойств.
- Методы доступа к компонентам приложения.
- Поддерживает интернационализацию.
- Загрузка файловых ресурсов обычным способом.



```
public interface Sim
{
    void calling();
    void data();
}
```

```
// Java Program to Illustrate Airtel Class

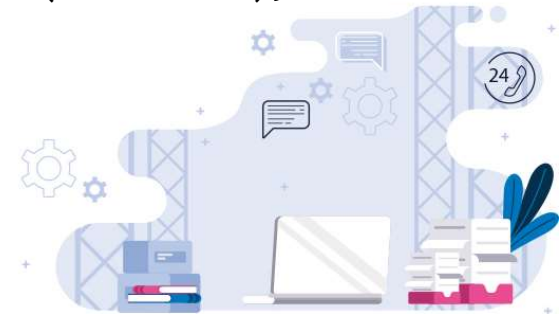
// Class
// Implementing Sim interface
public class Airtel implements Sim {

    @Override public void calling()
    {
        System.out.println("Airtel Calling");
    }

    @Override public void data()
    {
        System.out.println("Airtel Data");
    }
}
```

```
// Class
// Implementing Sim interface
public class Jio implements Sim{
    @Override
    public void calling() {
        System.out.println("Jio Calling");
    }

    @Override
    public void data() {
        System.out.println("Jio Data");
    }
}
```



```
// Class
public class Mobile {

    // Main driver method
    public static void main(String[] args)
    {

        // Creating instance of Sim interface
        // inside main() method
        // with reference to Jio class constructor
        // invocation
        Sim sim = new Jio();

        // Sim sim = new Airtel();

        sim.calling();
        sim.data();
    }

    Sim sim = new Vodafone();
```



# XML

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
https://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="sim" class="Jio"></bean>

</beans>
```



```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Mobile {
    public static void main(String[] args) {
        // Using ApplicationContext to implement Spring IoC
        ApplicationContext applicationContext = new ClassPathXmlApplicationContext("beans.xml");

        // Get the bean
        Sim sim = applicationContext.getBean("sim", Sim.class);

        // Calling the methods
        sim.calling();
        sim.data();
    }
}
```





# ПРОБЛЕМА

```
class ClassicalMusic {  
    // ... Код для доступа к классической музыке ...  
}  
  
class MusicPlayer {  
    private ClassicalMusic classicalMusic;  
  
    public void playMusic() {  
        classicalMusic = new ClassicalMusic();  
  
        // ... Код для воспроизведения музыки ...  
    }  
}
```

Проблема: в любом коде появляются зависимости



```
class ClassicalMusic {  
    // ... Код для доступа к классической музыке ...  
}  
  
class MusicPlayer {  
    private ClassicalMusic classicalMusic;  
  
    public void playMusic() {  
        classicalMusic = new ClassicalMusic();  
  
        // ... Код для воспроизведения музыки ...  
    }  
}
```

"Сильная зависимость"

**Проблема #1: MusicPlayer сильно зависит от ClassicalMusic**  
**Класс MusicPlayer "заточен" на работу только с ClassicalMusic**

**Решение: Использовать интерфейс (или абстрактный класс),  
который бы обобщал различные музыкальные жанры**





# Решение

```
interface Music {  
    // ... код, который необходим для доступа к любому жанру музыки ...  
}  
  
class ClassicalMusic implements Music {  
    // ... Код для доступа к классической музыке ...  
}  
  
class RockMusic implements Music {  
    // ... Код для доступа к рок музыке ...  
}  
  
class MusicPlayer {  
    private Music music;  
  
    public void playMusic() {  
        music = new ClassicalMusic();  
        // или  
        music = new RockMusic();  
  
        // ... Код для воспроизведения музыки ...  
    }  
}
```



```
class ClassicalMusic implements Music {  
    // ... Код для доступа к классической музыке ...  
}  
  
class RockMusic implements Music {  
    // ... Код для доступа к рок музыке ...  
}  
  
class MusicPlayer {  
    private Music music;  
  
    public void playMusic() {  
        music = new ClassicalMusic();  
        // или  
        music = new RockMusic();  
        // ... Код для воспроизведения музыки ...  
    }  
}
```

"Слабая зависимость"

**Проблема #2: Объекты создаются вручную.**

**Мы хотим вынести эти детали в конфигурационный файл, а не лезть каждый раз в код (и перекомпилировать его) для того, чтобы поменять объект**




# Решение

Решение использовать Spring Framework, который сам создаст необходимые объекты (бины), согласно конфигурационному файлу.



```
interface Music {  
    // ... код, который необходим для доступа к любому жанру музыки ...  
}  
  
class ClassicalMusic implements Music {  
    // ... Код для доступа к классической музыке ...  
}  
  
class RockMusic implements Music {  
    // ... Код для доступа к рок музыке ...  
}  
  
class MusicPlayer {  
    private Music music;  
  
    public void playMusic() {  
        music = new ClassicalMusic();  
        // или  
        music = new RockMusic();  
  
        // ... Код для воспроизведения музыки ...  
    }  
}
```



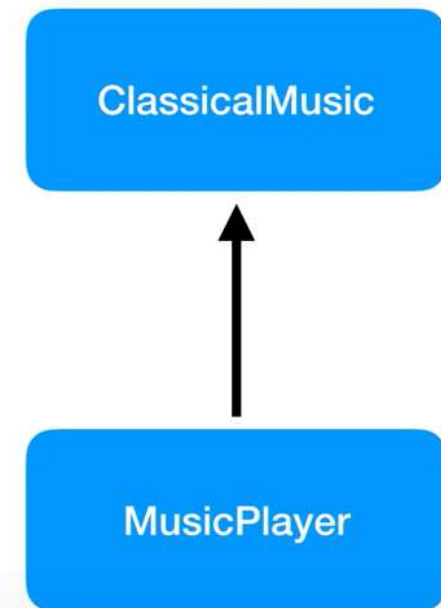
**Проблема #3: MusicPlayer сам создает свои зависимости.  
Это архитектурно неправильно - противоречит принципу IoC**



**Решение: Использовать принцип IoC**



```
interface Music {  
    // ... код, который необходим для доступа к любому жанру музыки ...  
}  
  
class ClassicalMusic implements Music {  
    // ... Код для доступа к классической музыке ...  
}  
  
class RockMusic implements Music {  
    // ... Код для доступа к рок музыке ...  
}  
  
class MusicPlayer {  
    private Music music;  
  
    public void playMusic() {  
        music = new ClassicalMusic();  
        // или  
        music = new RockMusic();  
  
        // ... Код для воспроизведения музыки ...  
    }  
}
```



- **MusicPlayer зависит от ClassicalMusic**
- **MusicPlayer сам создает объект ClassicalMusic**
- **Вместо этого мы хотим передавать объект ClassicalMusic внутрь MusicPlayer - это и называется инверсией управления (IoC)**



# Inversion of Control (IoC)

```
interface Music {  
    // ... код, который необходим для доступа к любому жанру музыки ...  
}  
  
class ClassicalMusic implements Music {  
    // ... код для доступа к классической музыке ...  
}  
  
class RockMusic implements Music {  
    // ... код для доступа к рок музыке ...  
}  
  
class MusicPlayer {  
    private Music music;  
  
    public void playMusic() {  
        music = new ClassicalMusic();  
        // или  
        music = new RockMusic();  
  
        // ... код для воспроизведения музыки ...  
    }  
}
```



Программируем на уровне  
интерфейсов - хороший  
архитектурный паттерн



```
class MediaPlayer {  
    private Music music;  
  
    // Зависимость внедряется извне (IoC)  
    public MediaPlayer(Music music) {  
        this.music = music;  
    }  
  
    public void playMusic() {  
        // Больше не создаем объекты!  
  
        // ... Код для воспроизведения музыки ...  
    }  
}
```

**Объект, который мы хотим внедрить в MediaPlayer  
необходимо где-то создавать.**

```
class UseMediaPlayer {  
    public static void main(String[] args) {  
        MediaPlayer musicPlayer = new MediaPlayer(new ClassicalMusic());  
    }  
}
```

**Объект, который мы хотим внедрить в MediaPlayer необходимо где-то создавать.**

```
class UseMediaPlayer {  
    public static void main(String[] args) {  
        MediaPlayer musicPlayer = new MediaPlayer(new ClassicalMusic());  
    }  
}
```

- Эту проблему мы решим на следующем уроке с помощью Внедрения зависимостей (Dependency Injection)
  - Этой задачей тоже занимается Spring Framework





# Сравнение конфигураций

- XML-based - классика, тьма документации с ней
- Java-based
  - Многие ошибки перешли в compile-time
  - Для библиотечных классов (@Service не поставить)
  - Нет зависимости в доменных классах от Spring
  - Когда несколько бинов (объектов) на класс (DataSource)
- Annotation-based – когда на класс сервиса один бин



# loc

```
class MusicPlayer {  
    private Music music;  
  
    // Зависимость внедряется извне (IoC)  
    public MusicPlayer(Music music) {  
        this.music = music;  
    }  
  
    public void playMusic() {  
        // Больше не создаем объекты!  
  
        // ... Код для воспроизведения музыки ...  
    }  
}
```



Решили некоторые проблемы, связанные с зависимостями  
С помощью Inversion of Control (IoC)

```
Music music = context.getBean( name: "musicBean", Music.class);  
MusicPlayer musicPlayer = new MusicPlayer(music);
```

**Но мы все еще внедряем зависимость music вручную  
И вручную создаем объект musicPlayer**

## Шаги

- Создаем Java - классы (будущие бины)
- Создаем и связываем бины с помощью Spring (аннотации, XML или Java код)
- При использовании, все объекты (бины) берутся из контейнера Spring

# Способы внедрения зависимостей

- Через **конструктор**
- Через **setter**
- Есть множество конфигураций того, как внедрять (**scope**, **factory method** и.т.д.)
- Можно внедрять через **XML**, **аннотации** или **Java - код**
- Процесс внедрения можно автоматизировать (**Autowiring**)

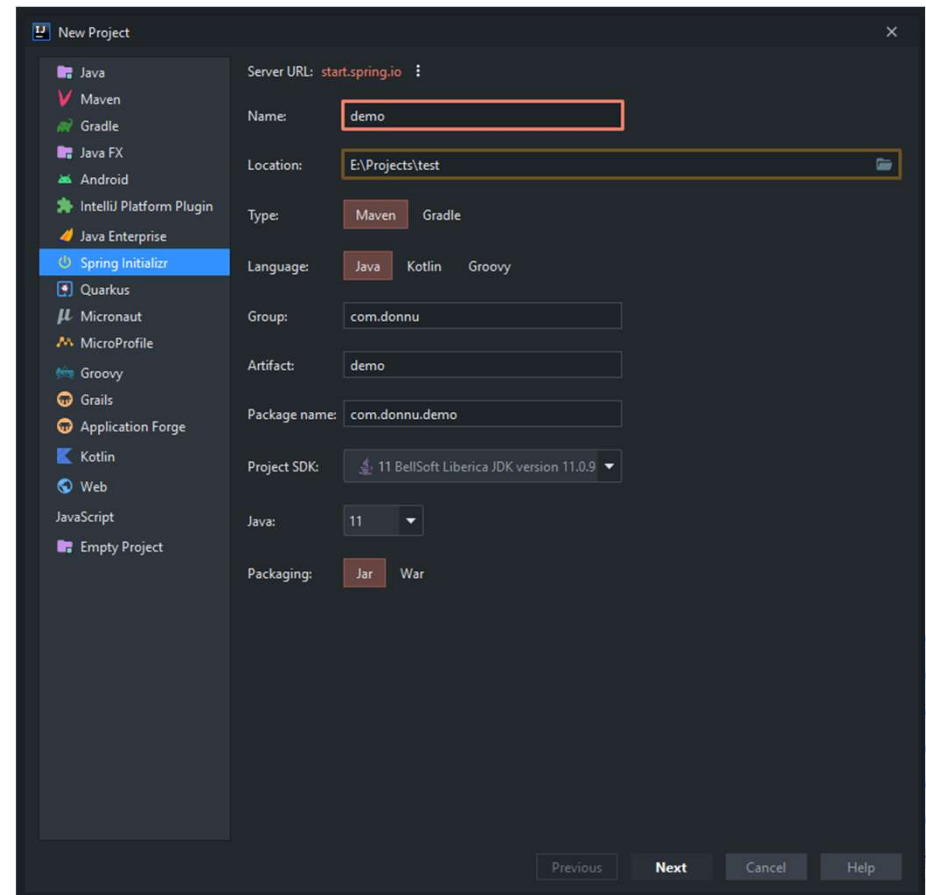
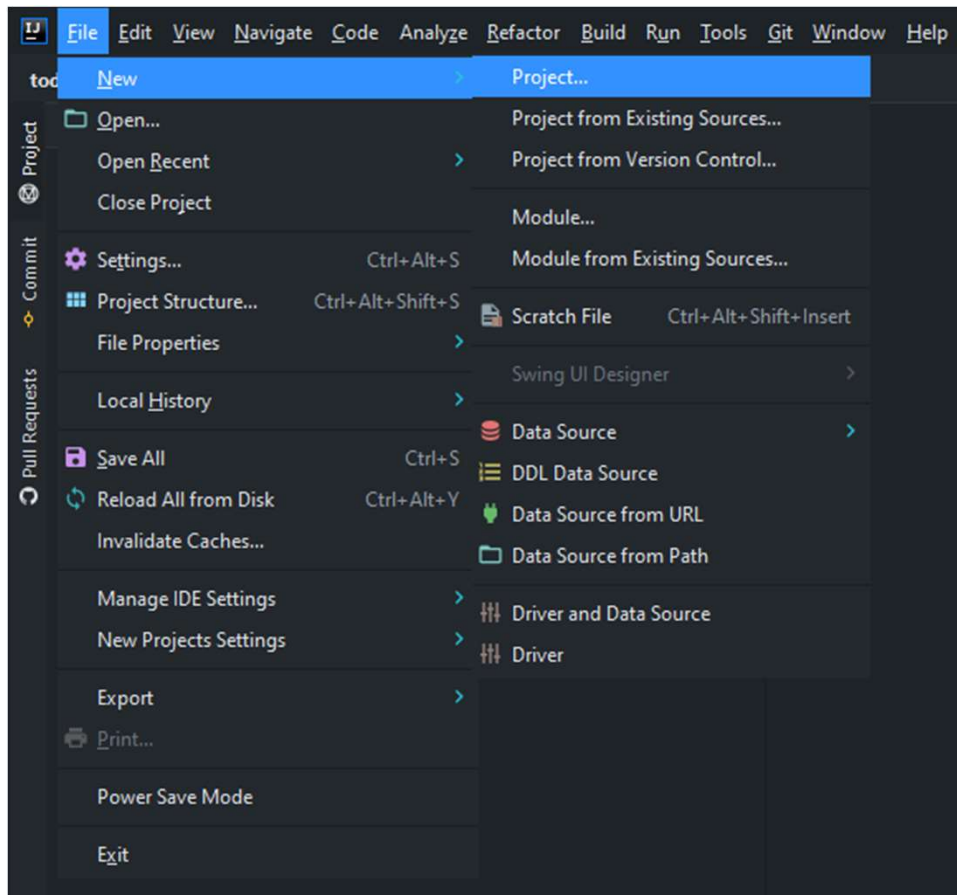


# Что такое внедрение зависимостей:

Внедрение зависимостей — это основная функциональность, предоставляемая [Spring](#) IOC (инверсия управления). Модуль Spring-Core отвечает за внедрение зависимостей с помощью методов Constructor или Setter. Принцип проектирования Inversion of Control подчеркивает сохранение независимости классов Java друг от друга, а контейнер освобождает их от создания и обслуживания объектов. Эти классы, управляемые [Spring](#), должны соответствовать стандартному определению Java-Bean. Внедрение зависимостей в [Spring](#) также обеспечивает слабую связь между классами.



# Создание проекта в IntelliJ IDEA



# Создание проекта в IntelliJ IDEA

