17.图书系统项目（一）

本节目标

- 项目演示
- 项目背景与知识点介绍
- 项目前置知识讲解

1. 项目演示

课上演示项目的主要功能

2. 项目背景与知识点介绍

2.1 为什么做这个项目

到目前为止我们学习过了Java的基础语法，包括数据类型，运算符，方法、类和对象、封装继承多态、抽象类接口、异常等等知识点。我们需要将这些知识点全部串起来，为该阶段学习划上一个圆满的句号。希望在学习完该项目后，同学们可以举一反三，做出更多的相同或者类似的项目来巩固这部分知识点。

2.2 目标

- 掌握项目从0到1的过程
- 掌握基本的类与对象的知识如：封装、继承、多态
- 掌握项目的调试
- 了解单例模式、代理模式、工厂模式
- 了解使用JSON序列化数据

3. 项目前置知识讲解

3.1 单例模式

3.1.1 基本概念

单例模式是一种设计模式，确保某一个类只有一个实例，并提供一个全局访问点。

这种模式常用于那些需要严格控制资源访问的场景，例如，数据库连接池、配置文件读取等。单例模式通过确保一个类只有一个实例来避免资源浪费和冲突。

单例模式的实现方式

单例模式的实现主要有两种方式：饿汉式和懒汉式。

1. 饿汉式（Eager Initialization）：在应用程序启动时就进行单例对象的初始化，无论是否会被使用。优点是不需要考虑多线程环境下的线程安全性，访问单例对象时不会引入额外的性能开销。缺点是可能会浪费系统资源，因为单例对象在应用程序启动时就被创建，如果一直未被使用，可能会占用内存。

```
1  public class Singleton {
2      private static Singleton instance = new Singleton();
3
4      private Singleton () {
5
6      }
7
8      public static Singleton getInstance() {
9          return instance;
10     }
11 }
```

2. 懒汉式（Lazy Initialization）：延迟加载，即单例对象在首次访问时才进行初始化。优点是节省了系统资源，只有在需要时才创建单例对象。缺点是在多线程环境下，可能会出现竞态条件，需要额外的线程安全措施来确保只创建一个实例。

```
1  public class Singleton {
2
3      private static Singleton instance;
4
5      private Singleton () {
6
7      }
8      public static Singleton getInstance() {
9          if (instance == null) {
10             instance = new Singleton();
11         }
12         return instance;
13     }
14 }
```

注意：

> - 该实现方式为线程不安全的，懒汉式的单例模式。
> - 后续课程当中会讲解线程安全的单例模式

我们在项目当中使用到的是第2种方式，事实上第2种方式还需要进一步优化，后续课程当中会给大家讲解到如何实现。

3.1.2 案例讲解

在任何一所学校当中：

- 校长只有一位(不算副校长)。
- 所有老师都应该能够方便的联系到校长
- 校长需要处理全校的比较重要的事务

我们需要保证校长只能有一位，此时的代码的设计如下：

```
public class Principal {
    private static Principal instance;
    private String name;

    // 构造函数设计为私有 防止实例化多个对象
    private Principal() {
        name = "王校长";
    }

    // 获取校长实例的公共方法
    public static Principal getInstance() {
        if (instance == null) {
            instance = new Principal();
        }
        return instance;
    }

    // 校长的一些方法
    public void announcement(String message) {
        System.out.println(name + "通知: " + message);
    }

    public void approveDocument(String file) {
        System.out.println(name + "审批通过: " + file);
    }
}
```

```java
public class School {
    public static void main(String[] args) {

        Principal principal1 = Principal.getInstance();
        principal1.announcement("全校明天开始放假3天");


        Principal principal2 = Principal.getInstance();
        principal2.approveDocument("食堂伙食改善文件");

        // 检查是否是同一个校长实例
        System.out.println("上述两个是同一位校长吗？ " + (principal1 == principal2));
    }
}
```

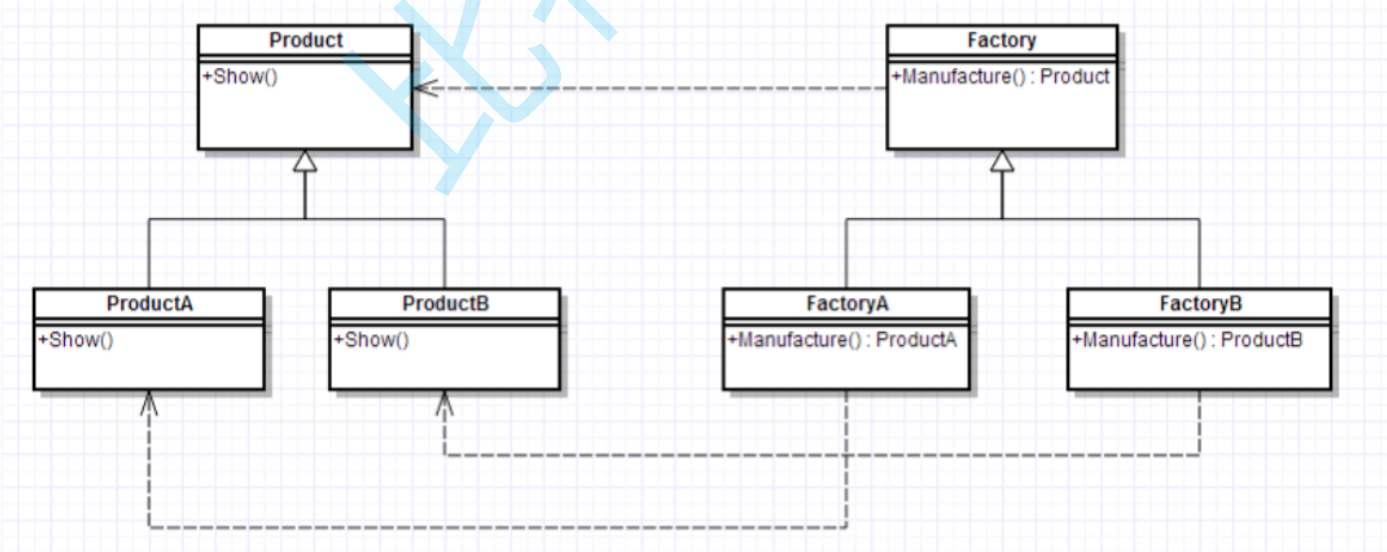
3.2 工厂模式

3.2.1 基本概念

工厂模式（Factory Pattern）是最常用的设计模式之一，它提供了一种创建对象的方式，使得创建对象的过程与使用对象的过程分离。

简单来说，工厂模式通过创建一个专门负责创建对象的工厂类，将对象的创建过程封装起来，以此达到解耦的目的。从而达到了解耦。

工厂模式分为：**简单工厂模式、工厂方法模式、抽象工厂模式。**我们这里主要讲解**工厂方法模式。**



组成（角色）	作用
抽象产品	工厂方法模式创建的所有类型的超级父类，该类型和具体业务有关，用于规范式中创建的方法对象具备的公共特征行为
具体产品	该类型实现了抽象产品 父类，是工厂方法模式中具体创建的实例对象
抽象工厂	整个工厂模式的核心角色，它与应用无关，主要在创建模式中规范和产品对应的标准化定义
具体工厂	实现了抽象工厂的具体工厂类，该类型是和应用直接交互的具体实现类，在应用，用于创建产品对象

3.2.2 案例讲解

我们需要创建1个管理员，2个普通用户对象。如果不使用工厂模式的情况下，我们的代码是这样完成的：

```
1  public abstract class User {
2      protected String name;  // 用户名
3      protected int userID;   // 用户ID
4
5      public User(String name, int userID) {
6          this.name = name;
7          this.userID = userID;
8      }
9  }
```

```
1  public class UserManagement {
2      public static void main(String[] args) {
3          // 直接创建管理员用户
4          AdminUser adminUser = new AdminUser("刘备", 1);
5          // 直接创建普通用户
6          NormalUser normalUser1 = new NormalUser("关羽", 2);
7
8          NormalUser normalUser2 = new NormalUser("张飞", 3);
9      }
10 }
```

缺点：

- 扩展性差，比如：我们要将NormalUser 改为 NormUser 那么所有创建对象的地方都得改动
- 代码重复：如果在多个地方需要创建用户，可能会导致代码重复。每次创建都是在new .....

使用工厂方法模式：

> 抽象产品：`User`
>
> 具体产品：`AdminUser` `NormalUser`

1. 创建工厂接口【抽象工厂】

```
public interface IUserFactory {
    User createUser(String name, int userID);
}
```

2. 实现具体工厂

```
public class AdminUserFactory implements IUserFactory {
    @Override
    public User createUser(String name, int userID) {
        return new AdminUser(name, userID);
    }
}

public class NormalUserFactory implements IUserFactory {
    @Override
    public User createUser(String name, int userID) {
        return new NormalUser(name, userID);
    }
}
```

3. 使用方式

```
IUserFactory adminFactory = new AdminUserFactory();
User adminUser = adminFactory.createUser("刘备", 1);

IUserFactory normalFactory = new NormalUserFactory();
User normalUser1 = normalFactory.createUser("关羽", 2);
User normalUser2 = normalUserFactory.createUser("张飞",3);
```

- 在工厂方法模式中，每种用户类型都有自己的工厂类。要添加新的用户类型，只需创建新的工厂类，无需修改现有代码。
- 代码不会重复，创建对象的过程，封装到了工厂当中
- 更加解耦

3.3 代理模式

3.3.1 基本概念

在代理模式（Proxy Pattern）中，一个类代表另一个类的功能，这种类型的设计模式属于结构型模式。

简单来说，代理模式就是通过代理对象来控制对实际对象的访问。代理对象在客户端和目标对象之间起到了中介的作用。

从分类上来说分为：

- **静态代理（Static Proxy）**
- **动态代理（Dynamic Proxy）**
- **CGLIB代理（Code Generation Library Proxy）**

我们重点来看静态代理。

简单通过例子理解：

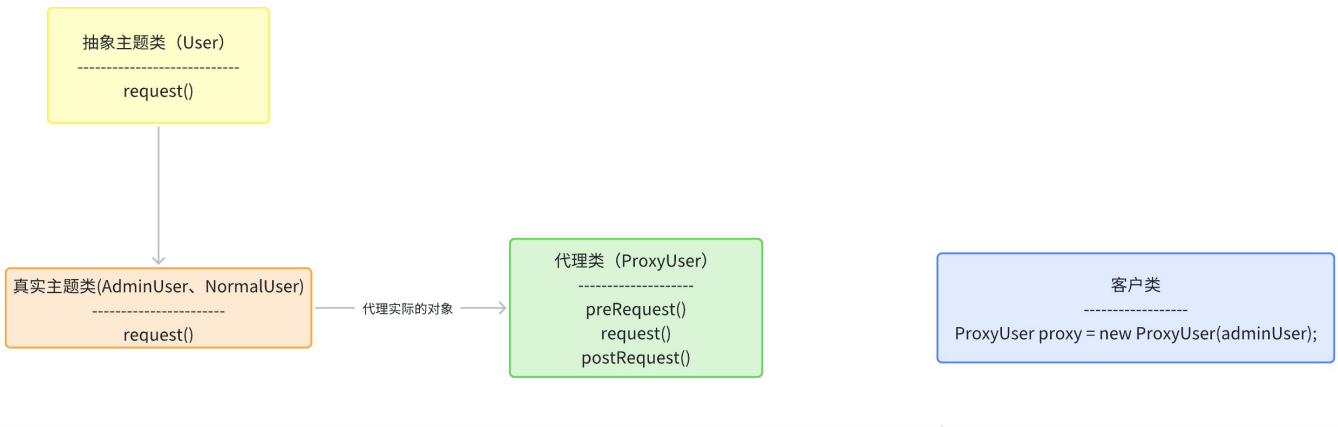
房东(目标对象)想把房子租出去，但是没有时间，此时只能要求中介(代理对象)来帮忙把房子租出去给客户(目标对象)。

3.3.2 代理模式结构图

角色：

1. 抽象主题类【业务接口类】：通过接口或者抽象类声明真实主题和代理对象实现的业务方法
2. 真实主题类【业务实现类】：实现抽象主题重点具体业务，是代理对象所代表的正式对象，也是最终需要引用的对象

3. 代理类：提供与真实主题相同的接口，其内部含有对真实主题的引用，他可以**访问、控制或扩展**真实主题的功能



1. 假设我们有一个User 类 如下：

```java
public abstract class User {
    protected String name;
    protected int userID;

    public User(String name, int userID) {
        this.name = name;
        this.userID = userID;
    }

    public abstract int display();
}
```

2. `NormalUser` 和 `AdminUser` 类的结构如下：

```java
public class NormalUser extends User {
    public NormalUser(String name, int userID) {
        super(name, userID);
    }

    public void borrowBook(String bookName) {
        System.out.println("普通用户 " + name + " 借阅了 " + bookName);
    }

    @Override
```

```
    public int display() {
        System.out.println("普通用户 " + name + " 的操作菜单:");
        System.out.println("1. 查找图书");
        System.out.println("2. 打印所有的图书");
        System.out.println("3. 退出系统");
        System.out.println("4. 借阅图书");
        System.out.println("5. 归还图书");
        System.out.println("6. 查看当前个人借阅情况");
        System.out.println("请选择你的操作: ");
        return scanner.nextInt();
    }
}

public class AdminUser extends User {
    public AdminUser(String name, int userID) {
        super(name, userID);
    }

    public void addBook(String bookName) {
        System.out.println("管理员 " + name + " 上架了 " + bookName);
    }

    @Override
    public int display() {
        System.out.println("管理员 " + name + " 的操作菜单:");
        System.out.println("1. 查找图书");
        System.out.println("2. 打印所有的图书");
        System.out.println("3. 退出系统");
        System.out.println("4. 上架图书");
        System.out.println("5. 修改图书");
        System.out.println("6. 下架图书");
        System.out.println("7. 统计借阅次数");
        System.out.println("8. 查看最后欢迎的前K本书");
        System.out.println("9. 查看库存状态");
        System.out.println("10. 按类别统计图书 ");
        System.out.println("11. 按作者统计图书 ");
        System.out.println("12. 检查超过一年未下架的图书");
        System.out.println("请选择你的操作: ");
        return scanner.nextInt();
    }
}
```

3. 代理类结构如下

```
public class ProxyUser {
```

```
    private User realUser;

    public ProxyUser(User user) {
        this.realUser = user;
    }

    //调用菜单
    public int display() {
        return realUser.display();
    }

    public void borrowBook(String bookName) {
        System.out.println("Proxy: 检查用户权限");
        if (realUser instanceof NormalUser) {
            ((NormalUser) realUser).borrowBook(bookName);
        } else if (realUser instanceof AdminUser) {
            System.out.println("您没有权限借阅书籍，请以普通用户的方式借阅书籍");
        }
    }

    public void addBook(String bookName) {
        System.out.println("Proxy: 检查书本状态");
        if (realUser instanceof NormalUser) {
            System.out.println("您没有权限上架书籍，请以管理员用户的方式上架书籍");
        } else if (realUser instanceof AdminUser) {
            ((AdminUser) realUser).addBook(bookName);
        }
    }
}
```

- 在该代理类中对borrowBook、returnBook方法提供了代理
- `ProxyUser` 为不同类型的 `User` 提供代理

4. 客户端代码

```
IUserFactory adminUserFactory = new AdminUserFactory();
User adminUser = adminUserFactory.createUser("刘备",1);

IUserFactory normalUserFactory = new NormalUserFactory();
User normalUser1 = normalUserFactory.createUser("关羽",2);
User normalUser2 = normalUserFactory.createUser("张飞",3);

ProxyUser proxyUserAdmin = new ProxyUser(adminUser);
ProxyUser proxyUserNormal1 = new ProxyUser(normalUser1);
ProxyUser proxyUserNormal2 = new ProxyUser(normalUser2);
```

```
11
12    proxyUserAdmin.addBook("Java编程思想");
13    proxyUserNormal1.borrowBook("设计模式");
14
```

此时我们通过代理可以在客户端和实际对象之间作为一个中间层，控制对实际对象的访问，同时控制对原始对象的访问权限，比如：检查调用者是否有足够的权限访问某个对象。

4. 关于jar包的使用

4.1 什么是jar包?

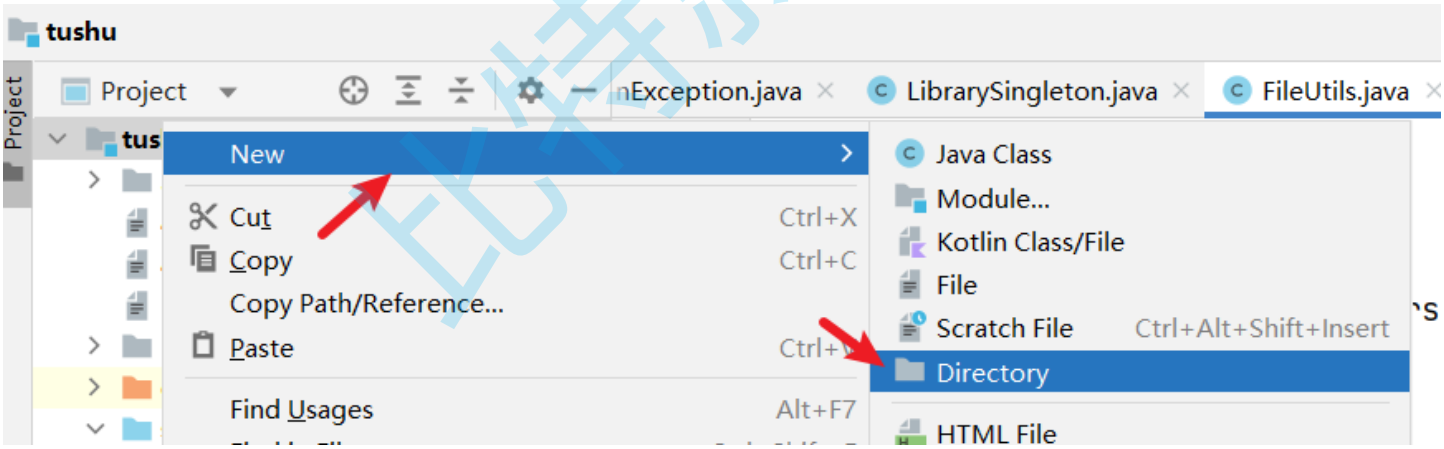
JAR（Java Archive）包是一种用于打包 Java 应用程序和相关组件的文件格式。它基本上是一个 ZIP 格式的压缩文件，但是专门用于 Java 平台。

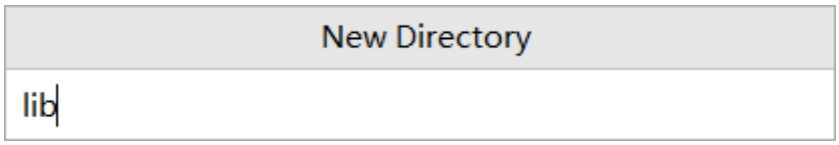
4.2 我们为什么使用jar包

实际上我们可以不使用jar包，但是本项目当中会遇到文件的操作，关于文件相关操作我们还没有学习，所以，我们提前写好封装到了jar包当中，我们直接通过配置就可以使用了。

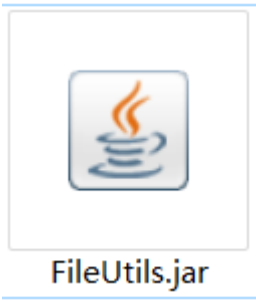
4.3 项目引入jar包

1. 打开项目，创建一个文件夹为：lib



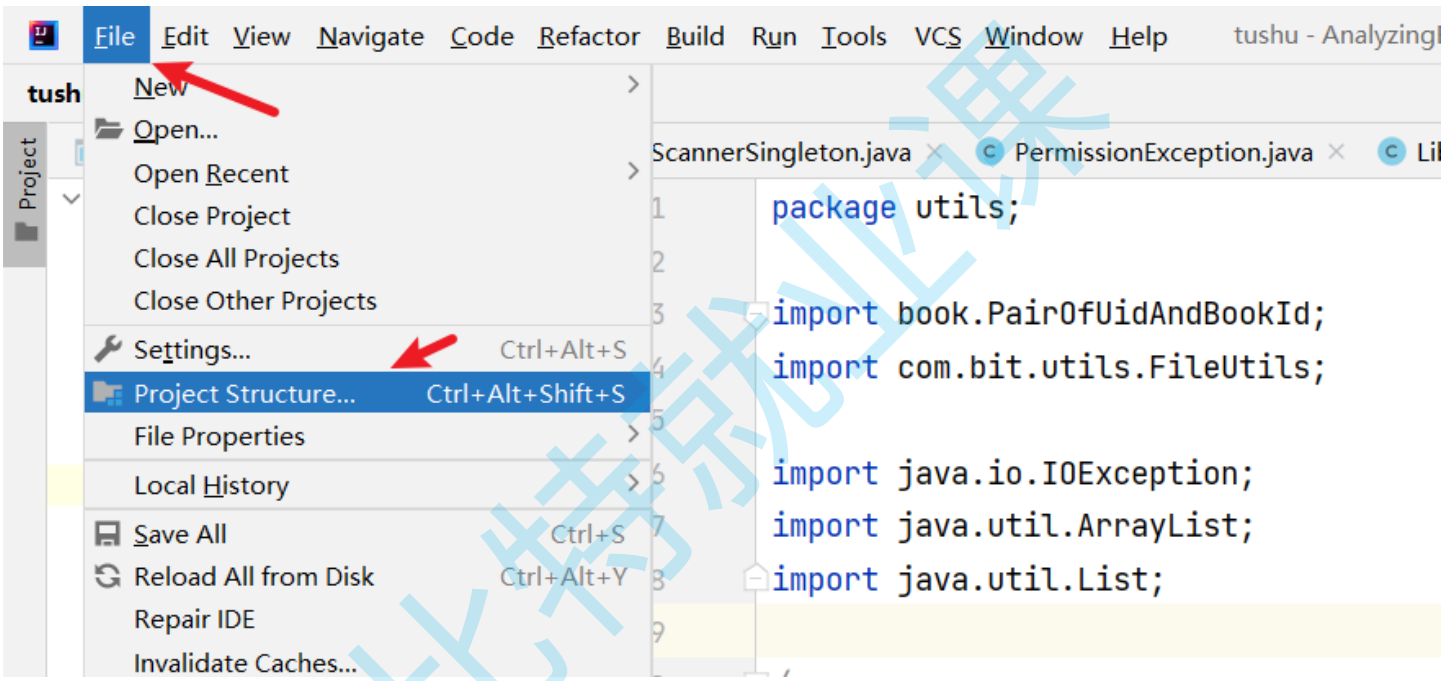


2. 把项目资料中的 `FileUtils.jar` 包放到该目录下即可

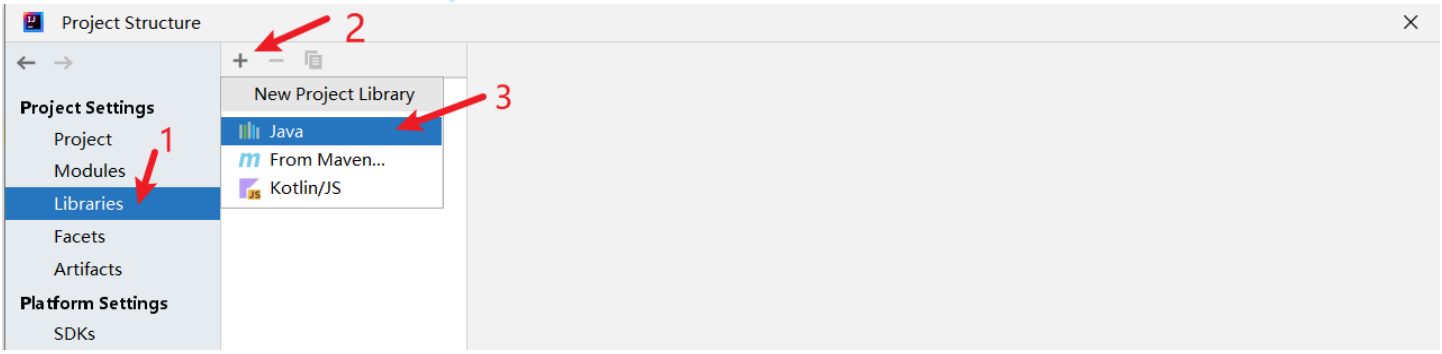


具体步骤如下：

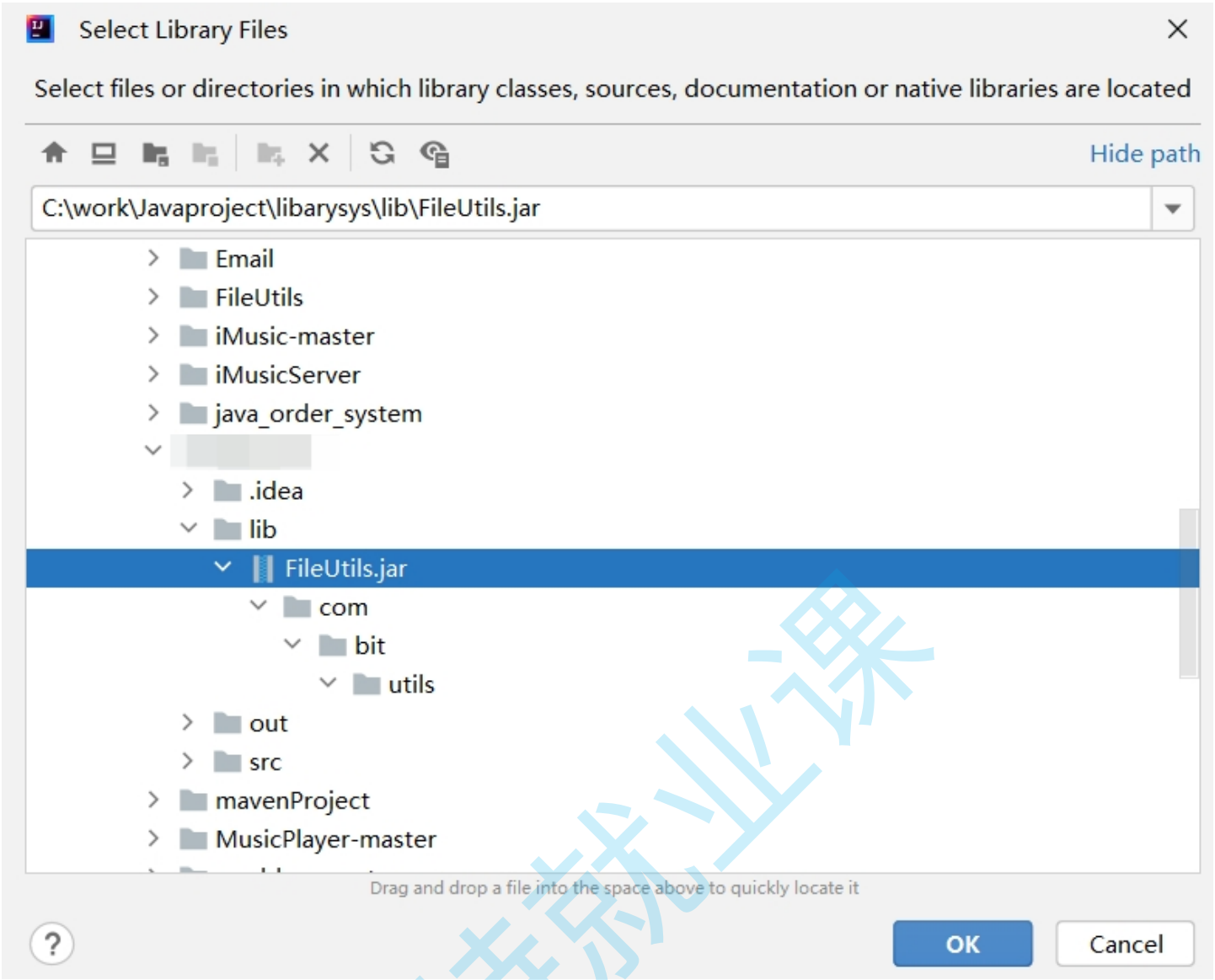
- 点击File，选择 "Project Stru....."

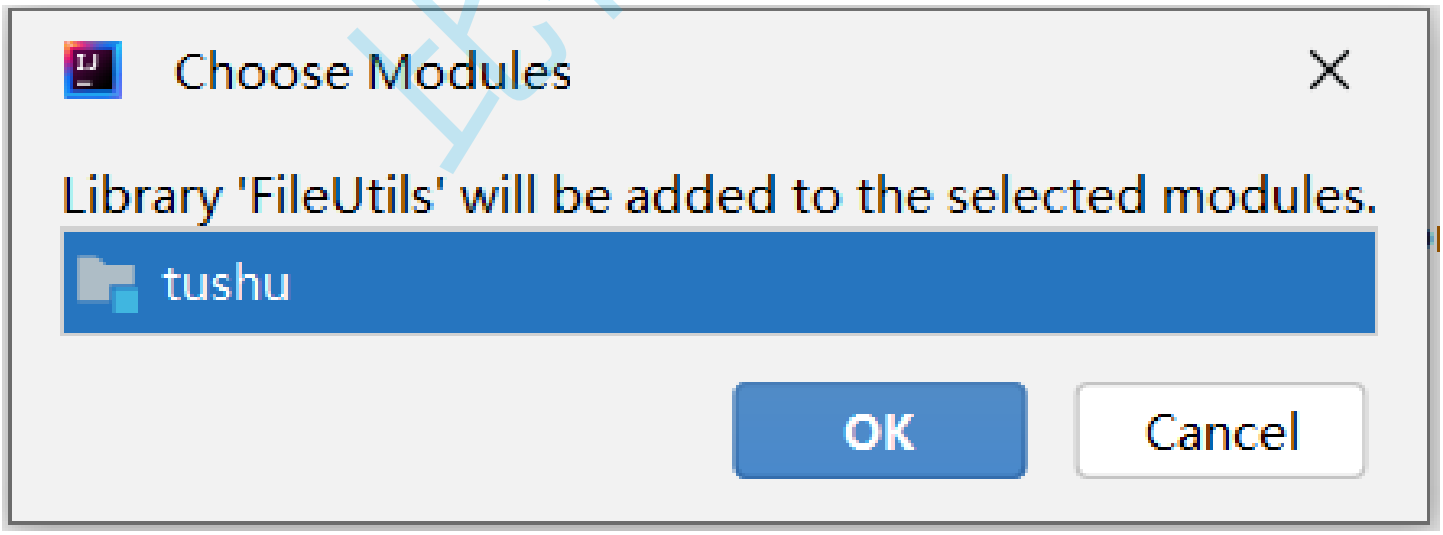


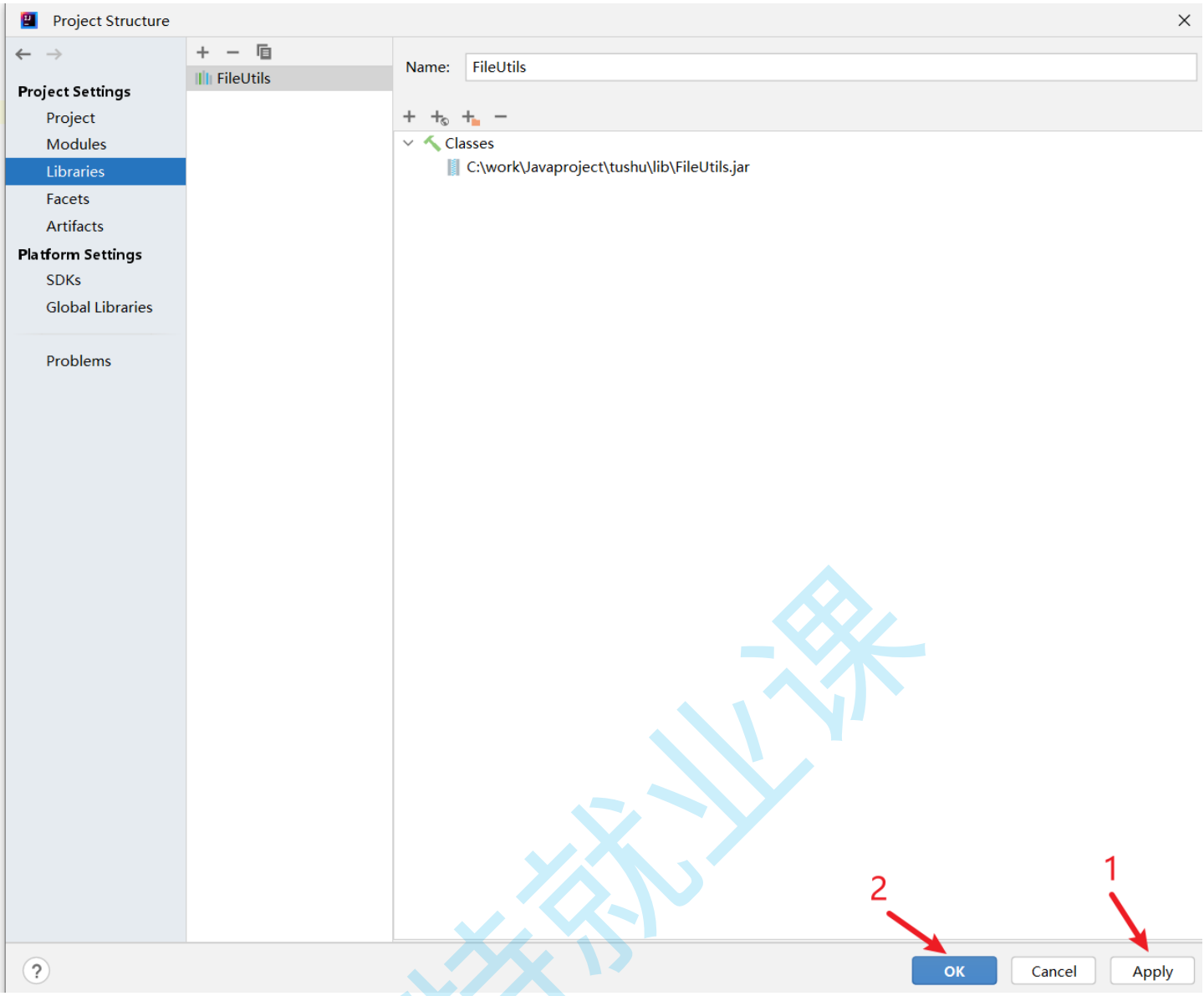
- 在左侧面板中，选择 "Libraries"，点击 "+" 按钮，选择 "Java"



- 浏览并选择你刚刚创建的 JAR 文件

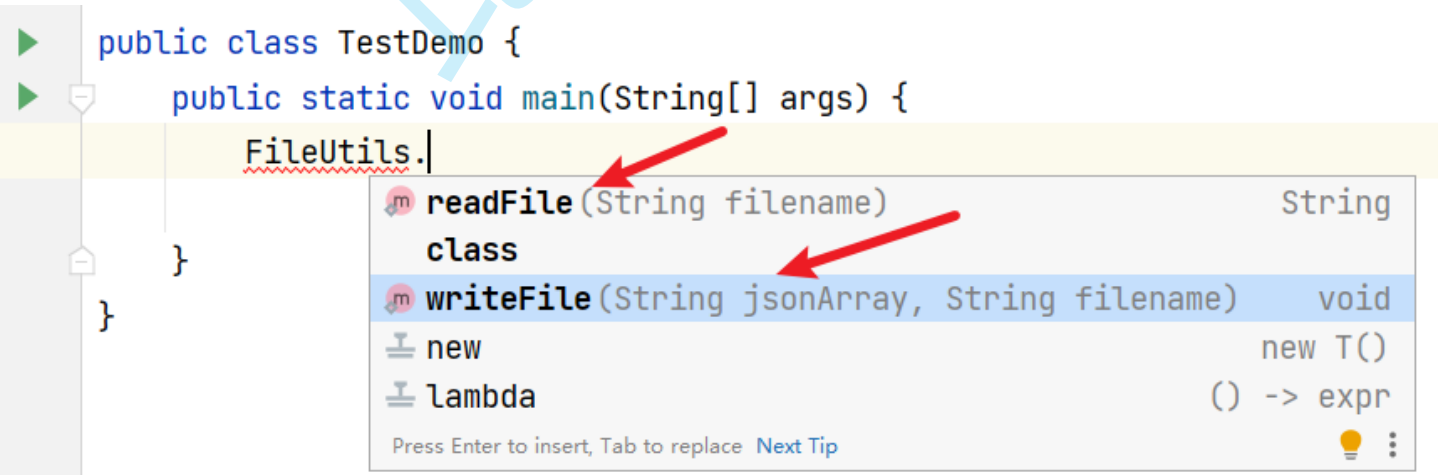

Select Library Files
Select files or directories in which library classes, sources, documentation or native libraries are located
Hide path
C:\work\Javaproject\libarysys\lib\FileUtils.jar
Email
FileUtils
iMusic-master
iMusicServer
java_order_system
.idea
lib
FileUtils.jar
com
bit
utils
out
src
mavenProject
MusicPlayer-master
Drag and drop a file into the space above to quickly locate it
OK
Cancel


Choose Modules
Library 'FileUtils' will be added to the selected modules.
tushu
OK
Cancel



测试导入是否成功：

如果能够通过点号能够找到这两个方法那么我们导入jar包就没有问题了！



完