day16 单例模式、多例模式、枚举、工厂模式

教学目标

- [ ] 能够说出单例设计模式的好处
- [ ] 能够说出多例模式的好处
- [ ] 能够定义枚举
- [ ] 能够使用工厂模式编写java程序

第一章 单例设计模式

正常情况下一个类可以创建多个对象

```
public static void main(String[] args) {
    // 正常情况下一个类可以创建多个对象
    Person p1 = new Person();
    Person p2 = new Person();
    Person p3 = new Person();
}
```

1.1 单例设计模式的作用

单例模式，是一种常用的软件设计模式。通过单例模式可以保证系统中，应用该模式的这个类只有一个实例。即一个类只有一个对象实例。

1.2 单例设计模式实现步骤

1. 将构造方法私有化，使其不能在类的外部通过new关键字实例化该类对象。
2. 在该类内部产生一个唯一的实例化对象，并且将其封装为private static类型的成员变量。
3. 定义一个静态方法返回这个唯一对象。

1.3 单例设计模式的类型

根据实例化对象的时机单例设计模式又分为以下两种:

1. 饿汉单例设计模式
2. 懒汉单例设计模式

1.4 饿汉单例设计模式

饿汉单例设计模式就是使用类的时候已经将对象创建完毕，不管以后会不会使用到该实例化对象，先创建了再说。很着急的样子，故被称为“饿汉模式”。

代码如下:

```
public class Singleton {
    // 1.将构造方法私有化，使其不能在类的外部通过new关键字实例化该类对象。
    private Singleton() {}

    // 2.在该类内部产生一个唯一的实例化对象，并且将其封装为private static类型的成员变量。
    private static final Singleton instance = new Singleton();

    // 3.定义一个静态方法返回这个唯一对象。
    public static Singleton getInstance() {
        return instance;
    }
}
```

1.5 懒汉单例设计模式

懒汉单例设计模式就是调用getInstance()方法时实例才被创建，先不急着实例化出对象，等要用的时候才例化出对象。不着急，故称为“懒汉模式”。

代码如下：

```
public class Singleton {

    // 2.在该类内部产生一个唯一的实例化对象，并且将其封装为private static类型的成员变量。
    private static Singleton instance;

    // 1.将构造方法私有化，使其不能在类的外部通过new关键字实例化该类对象。
    private Singleton() {}

    // 3.定义一个静态方法返回这个唯一对象。要用的时候才例化出对象
    public static synchronized Singleton getInstance() {
        if(instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

> 注意：懒汉单例设计模式在多线程环境下可能会实例化出多个对象，不能保证单例的状态。我们在学习完多线程的时候还会再讲解如何解决这个问题。

1.7 小结

单例模式可以保证系统中一个类只有一个对象实例。

实现单例模式的步骤：

1. 将构造方法私有化，使其不能在类的外部通过new关键字实例化该类对象。
2. 在该类内部产生一个唯一的实例化对象，并且将其封装为private static类型的成员变量。
3. 定义一个静态方法返回这个唯一对象。

第二章 多例设计模式

一般情况下一个类可以创建多个对象

```
public static void main(String[] args) {
    // 正常情况下一个类可以创建多个对象
    Person p1 = new User();
    Person p2 = new User();
    Person p3 = new User();
}
```

2.1.多例设计模式的作用

多例模式，是一种常用的软件设计模式。通过多例模式可以保证系统中，应用该模式的类有固定数量的实例。多例类要自我创建并管理自己的实例，还要向外界提供获取本类实例的方法。

2.2.实现步骤

1.创建一个类, 将构造方法私有化，使其不能在类的外部通过new关键字实例化该类对象。

2.在类中定义该类被创建的总数量

3.在类中定义存放类实例的list集合

4.在类中提供静态代码块,在静态代码块中创建类的实例

5.提供获取类实例的静态方法

2.3.实现代码如下:

```
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
public class Multition {
    // 定义该类被创建的总数量
    private static final int maxCount = 3;
    // 定义存放类实例的list集合
    private static List instanceList = new ArrayList();
    // 构造方法私有化,不允许外界创建本类对象
    private Multition() {
    }
    static {
        // 创建本类的多个实例,并存放到list集合中
        for (int i = 0; i < maxCount; i++) {
            Multition multition = new Multition();
            instanceList.add(multition);
        }
    }
    // 给外界提供一个获取类对象的方法
    public static Multition getMultition(){
        Random random = new Random();
        // 生成一个随机数
        int i = random.nextInt(maxCount);
        // 从list集合中随机取出一个进行使用
        return (Multition)instanceList.get(i);
    }
}
```

2.4.测试结果:

```java
public static void main(String[] args) {
    // 编写一个循环从中获取类对象
    for (int i = 0; i < 10; i++) {
        Multition multition = Multition.getMultition();
        System.out.println(multition);
    }
}
```

```
D:\software\java\jdk\jdk1.8.0_45_x64\bin\java ...
com.itheima.Multition@60e53b93
com.itheima.Multition@5e2de80c
com.itheima.Multition@60e53b93
com.itheima.Multition@5e2de80c
com.itheima.Multition@60e53b93
com.itheima.Multition@1d44bcfa
com.itheima.Multition@60e53b93
com.itheima.Multition@5e2de80c
com.itheima.Multition@5e2de80c
com.itheima.Multition@1d44bcfa
```

2.5.多例模式小结:

多例模式可以保证系统中一个类有固定个数的实例, 在实现需求的基础上, 能够提高实例的复用性.

实现多例模式的步骤:

1. 创建一个类, 将构造方法私有化，使其不能在类的外部通过new关键字实例化该类对象。
2. 在类中定义该类被创建的总数量
3. 在类中定义存放类实例的list集合
4. 在类中提供静态代码块,在静态代码块中创建类的实例
5. 提供获取类实例的静态方法

第三章 枚举

3.1 不使用枚举存在的问题

假设我们要定义一个人类，人类中包含姓名和性别。通常会将性别定义成字符串类型，效果如下:

```java
public class Person {
    private String name;
    private String sex;

    public Person() {
    }

    public Person(String name, String sex) {
        this.name = name;
```

```
        this.sex = sex;
    }

    // 省略get/set/toString方法
}
```

```
public class Demo01 {
    public static void main(String[] args) {
        Person p1 = new Person("张三", "男");
        Person p2 = new Person("张三", "abc"); // 因为性别是字符串,所以我们可以传入任意
字符串
    }
}
```

不使用枚举存在的问题：可以给性别传入任意的字符串，导致性别是非法的数据，不安全。

3.2 枚举的作用与应用场景

枚举的作用：一个方法接收的参数是固定范围之内的时候，那么即可使用枚举。

3.3 枚举的基本语法

3.3.1 枚举的概念

枚举是一种特殊类。枚举是有固定实例个数的类型，我们可以把枚举理解成有固定个数实例的多例模式。

3.3.2 定义枚举的格式

```
enum 枚举名 {
    第一行都是罗列枚举实例,这些枚举实例直接写大写名字即可。
}
```

3.3.3 入门案例

1. 定义枚举：BOY表示男，GIRL表示女

```
enum Sex {
    BOY, GIRL; // 男，女
}
```

2. Perosn中的性别有String类型改为Sex枚举类型

```
public class Person {
    private String name;
    private Sex sex;

    public Person() {
    }

    public Person(String name, Sex sex) {
        this.name = name;
        this.sex = sex;
    }
    // 省略get/set/toString方法
}
```

3. 使用是只能传入枚举中的固定值

```
public class Demo02 {
    public static void main(String[] args) {
        Person p1 = new Person("张三", Sex.BOY);
        Person p2 = new Person("张三", Sex.GIRL);
        Person p3 = new Person("张三", "abc");
    }
}
```

3.3.4 枚举的其他内容

枚举的本质是一个类，我们刚才定义的Sex枚举最终效果如下：

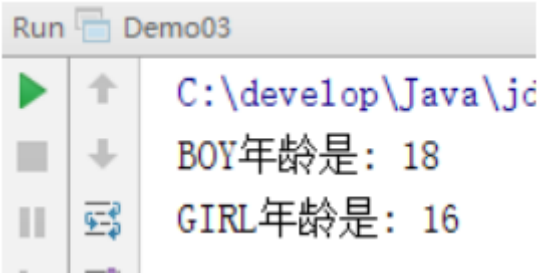
```
enum Sex {
    BOY, GIRL; // 男，女
}

// 枚举的本质是一个类，我们刚才定义的Sex枚举相当于下面的类
final class SEX extends java.lang.Enum<SEX> {
    public static final SEX BOY = new SEX();
    public static final SEX GIRL = new SEX();
    public static SEX[] values();
    public static SEX valueOf(java.lang.String);
    static {};
}
```

枚举的本质是一个类，所以枚举中还可以有成员变量，成员方法等。

```
public enum Sex {
    BOY(18), GIRL(16);

    public int age;

    Sex(int age) {
        this.age = age;
    }

    public void showAge() {
        System.out.println("年龄是: " + age);
    }
}
```

```
public class Demo03 {
    public static void main(String[] args) {
        Person p1 = new Person("张三", Sex.BOY);
        Person p2 = new Person("张三", Sex.GIRL);

        Sex.BOY.showAge();
        Sex.GIRL.showAge();
    }
}
```

运行效果:



3.4 应用场景

3.5 枚举的应用

枚举的作用：枚举通常可以用于做信息的分类，如性别，方向，季度等。

枚举表示性别:

```
public enum Sex {
    MAIL, FEMAIL;
}
```

枚举表示方向:

```
public enum Orientation {
    UP, RIGHT, DOWN, LEFT;
}
```

枚举表示季度

```
public enum Season {
    SPRING, SUMMER, AUTUMN, WINTER;
}
```

3.6 小结

- 枚举类在第一行罗列若干个枚举对象。（多例）
- 第一行都是常量，存储的是枚举类的对象。
- 枚举是不能在外部创建对象的，枚举的构造器默认是私有的。
- 枚举通常用于做信息的标志和分类。

第四章 工厂设计模式

4.1.工厂模式概述:

工厂模式（Factory Pattern）是 Java 中最常用的设计模式之一。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。之前我们创建类对象时, 都是使用new 对象的形式创建, 除new对象方式以外, 工厂模式也可以创建对象.

4.2.工厂模式作用:

解决类与类之间的耦合问题

4.3.工厂模式实现步骤:

1. 编写一个Car接口, 提供run方法
2. 编写一个Falali类实现Car接口,重写run方法
3. 编写一个Benchi类实现Car接口
4. 提供一个CarFactory(汽车工厂),用于生产汽车对象
5. 定义CarFactoryTest测试汽车工厂

4.4.工厂模式实现代码:

1.编写一个Car接口, 提供run方法

```
public interface Car {
    public void run();
}
```

2.编写一个Falali类实现Car接口,重写run方法

```
public class Falali implements Car {
    @Override
    public void run() {
        System.out.println("法拉利以每小时500公里的速度在奔跑.....");
    }
}
```

3.编写一个Benchi类实现Car接口

```
public class Benchi implements Car {
    @Override
    public void run() {
        System.out.println("奔驰汽车以每秒1米的速度在挪动.....");
    }
}
```

4.提供一个CarFactory(汽车工厂),用于生产汽车对象

```
public class CarFactory {
    /**
     * @param id : 车的标识
     *           benchi : 代表需要创建Benchi类对象
     *           falali : 代表需要创建Falali类对象
     *           如果传入的车标识不正确,代表当前工厂生成不了当前车对象,则返回null
     * @return
     */
    public Car createCar(String id){
        if("falali".equals(id)){
            return new Falali();
        }else if("benchi".equals(id)){
            return new Benchi();
        }
        return null;
    }
}
```

5.定义CarFactoryTest测试汽车工厂

```
public class CarFactoryTest {
    public static void main(String[] args) {
        CarFactory carFactory = new CarFactory();
        Car benchi = carFactory.createCar("benchi");
        benchi.run();
        Car falali = carFactory.createCar("falali");
        falali.run();
    }
}
```

4.5.工厂模式小结

工厂模式的存在可以改变创建类的方式,解决类与类之间的耦合.

实现步骤:

1. 编写一个Car接口, 提供run方法
2. 编写一个Falali类实现Car接口,重写run方法
3. 编写一个Benchi类实现Car接口
4. 提供一个CarFactory(汽车工厂),用于生产汽车对象
5. 定义CarFactoryTest测试汽车工厂

第五章 复习SE

复习SE核心内容