1.接口

1.1黑马信息管理系统集合改进 (应用)

- 使用数组容器的弊端
  1. 容器长度是固定的，不能根据添加功能自动增长
  2. 没有提供用于赠删改查的方法
- 优化步骤
  1. 创建新的StudentDao类，OtherStudentDao
  2. 创建ArrayList集合容器对象
  3. OtherStudentDao中的方法声明，需要跟StudentDao保持一致

     注意：如果不一致，StudentService中的代码就需要进行修改
  4. 完善方法（添加、删除、修改、查看）
  5. 替换StudentService中的Dao对象
- 代码实现

  OtherStudentDao类

```
public class OtherStudentDao {
    // 集合容器
    private static ArrayList<Student> stus = new ArrayList<>();

    static {
        Student stu1 = new Student("heima001","张三","23","1999-11-11");
        Student stu2 = new Student("heima002","李四","24","2000-11-11");

        stus.add(stu1);
        stus.add(stu2);
    }

    // 添加学生方法
    public boolean addStudent(Student stu) {
       stus.add(stu);
       return true;
    }

    // 查看学生方法
    public Student[] findAllStudent() {

        Student[] students = new Student[stus.size()];

        for (int i = 0; i < students.length; i++) {
            students[i] = stus.get(i);
        }


        return students;
```

```
    }

    public void deleteStudentById(String delId) {
        // 1. 查找id在容器中所在的索引位置
        int index = getIndex(delId);
        stus.remove(index);
    }

    public int getIndex(String id){
        int index = -1;
        for (int i = 0; i < stus.size(); i++) {
            Student stu = stus.get(i);
            if(stu != null && stu.getId().equals(id)){
                index = i;
                break;
            }
        }
        return index;
    }

    public void updateStudent(String updateId, Student newStu) {
        // 1. 查找updateId, 在容器中的索引位置
        int index = getIndex(updateId);
        stus.set(index, newStu);
    }
}
```

StudentService类

```
public class StudentService {
    // 创建StudentDao (库管)
     private OtherStudentDao studentDao = new OtherStudentDao();
    // 其他方法没有变化,此处省略...
}
```

1.2黑马信息管理系统抽取Dao (应用)

- 优化步骤
  1. 将方法向上抽取，抽取出一个父类 （ BaseStudentDao ）
  2. 方法的功能实现在父类中无法给出具体明确，定义为抽象方法
  3. 让两个类分别继承 BaseStudentDao ， 重写内部抽象方法
- 代码实现

  BaseStudentDao类

```
public abstract class BaseStudentDao {
    // 添加学生方法
    public abstract boolean addStudent(Student stu);
    // 查看学生方法
    public abstract Student[] findAllStudent();
    // 删除学生方法
    public abstract void deleteStudentById(String delId);
    // 根据id找索引方法
    public abstract int getIndex(String id);
    // 修改学生方法
    public abstract void updateStudent(String updateId, Student newStu);
}
```

StudentDao类

```
public class StudentDao extends BaseStudentDao {
  // 其他内容不变,此处省略
}
```

OtherStudentDao类

```
public class OtherStudentDao extends BaseStudentDao {
  // 其他内容不变,此处省略
}
```

1.3接口的概述（理解）

- 接口就是一种公共的规范标准，只要符合规范标准，大家都可以通用。
- Java中接口存在的两个意义
  1. 用来定义规范
  2. 用来做功能的拓展

1.4接口的特点（记忆）

- 接口用关键字interface修饰

```
public interface 接口名 {}
```

- 类实现接口用implements表示

```
public class 类名 implements 接口名 {}
```

- 接口不能实例化

  我们可以创建接口的实现类对象使用

- 接口的子类

  要么重写接口中的所有抽象方法

  要么子类也是抽象类

1.5接口的成员特点（记忆）

- 成员特点
  - 成员变量

    只能是常量 默认修饰符：public static final
  - 构造方法

    没有，因为接口主要是扩展功能的，而没有具体存在
  - 成员方法

    只能是抽象方法

    默认修饰符：public abstract

    关于接口中的方法，JDK8和JDK9中有一些新特性，后面再讲解
- 代码演示
  - 接口

```
public interface Inter {
    public static final int NUM = 10;

    public abstract void show();
}
```

  - 实现类

```
class InterImpl implements Inter{

    public void method(){
        // NUM = 20;
        System.out.println(NUM);
    }

    public void show(){

    }
}
```

  - 测试类

```
public class TestInterface {
    /*
        成员变量：只能是常量 系统会默认加入三个关键字
                    public static final
        构造方法：没有
        成员方法：只能是抽象方法，系统会默认加入两个关键字
                    public abstract
     */
    public static void main(String[] args) {
        System.out.println(Inter.NUM);
    }

}
```

1.6类和接口的关系（记忆）

- 类与类的关系

  继承关系，只能单继承，但是可以多层继承

- 类与接口的关系

  实现关系，可以单实现，也可以多实现，还可以在继承一个类的同时实现多个接口

- 接口与接口的关系

  继承关系，可以单继承，也可以多继承

1.7黑马信息管理系统使用接口改进 (应用)

- 实现步骤

  1. 将 BaseStudentDao 改进为一个接口
  2. 让 StudentDao 和 OtherStudentDao 去实现这个接口

- 代码实现

  BaseStudentDao接口

```
public interface BaseStudentDao {
    // 添加学生方法
    public abstract boolean addStudent(Student stu);
    // 查看学生方法
    public abstract Student[] findAllStudent();
    // 删除学生方法
    public abstract void deleteStudentById(String delId);
    // 根据id找索引方法
    public abstract int getIndex(String id);
    // 修改学生方法
    public abstract void updateStudent(String updateId, Student newStu);
}
```

  StudentDao类

```
public class StudentDao implements BaseStudentDao {
  // 其他内容不变,此处省略
}
```

OtherStudentDao类

```
public class OtherStudentDao implements BaseStudentDao {
  // 其他内容不变,此处省略
}
```

1.8黑马信息管理系统解耦合改进 (应用)

- 实现步骤
  1. 创建factory包，创建 StudentDaoFactory（工厂类）
  2. 提供 static 修改的 getStudentDao 方法，该方法用于创建StudentDao对象并返回
- 代码实现

  StudentDaoFactory类

```
public class StudentDaoFactory {
    public static OtherStudentDao getStudentDao(){
        return new OtherStudentDao();
    }
}
```

  StudentService类

```
public class StudentService {
    // 创建StudentDao (库管)
    // private OtherStudentDao studentDao = new OtherStudentDao();

    // 通过学生库管工厂类，获取库管对象
    private OtherStudentDao studentDao = StudentDaoFactory.getStudentDao();
}
```

2.接口组成更新

2.1接口组成更新概述【理解】

- 常量

  public static final
- 抽象方法

  public abstract
- 默认方法(Java 8)
- 静态方法(Java 8)
- 私有方法(Java 9)

2.2接口中默认方法【应用】

- 格式

  public default 返回值类型 方法名(参数列表) { }

- 作用

  解决接口升级的问题

- 范例

```
public default void show3() {
}
```

- 注意事项
  - 默认方法不是抽象方法，所以不强制被重写。但是可以被重写，重写的时候去掉default关键字
  - public可以省略，default不能省略
  - 如果实现了多个接口，多个接口中存在相同的方法声明，子类就必须对该方法进行重写

2.3接口中静态方法【应用】

- 格式

  public static 返回值类型 方法名(参数列表) { }

- 范例

```
public static void show() {
}
```

- 注意事项
  - 静态方法只能通过接口名调用，不能通过实现类名或者对象名调用
  - public可以省略，static不能省略

2.4接口中私有方法【应用】

- 私有方法产生原因

  Java 9中新增了带方法体的私有方法，这其实在Java 8中就埋下了伏笔：Java 8允许在接口中定义带方法体的默认方法和静态方法。这样可能就会引发一个问题：当两个默认方法或者静态方法中包含一段相同的代码实现时，程序必然考虑将这段实现代码抽取成一个共性方法，而这个共性方法是不需要让别人使用的，因此用私有给隐藏起来，这就是Java 9增加私有方法的必然性

- 定义格式
  - 格式1

    private 返回值类型 方法名(参数列表) { }

  - 范例1

```
private void show() {
}
```

- 格式2

  private static 返回值类型 方法名(参数列表) { }

- 范例2

```
private static void method() {
}
```

- 注意事项
  - 默认方法可以调用私有的静态方法和非静态方法
  - 静态方法只能调用私有的静态方法

3.多态

3.1多态的概述（记忆）

- 什么是多态

  同一个对象，在不同时刻表现出来的不同形态

- 多态的前提
  - 要有继承或实现关系
  - 要有方法的重写
  - 要有父类引用指向子类对象
- 代码演示

```
class Animal {
    public void eat(){
        System.out.println("动物吃饭");
    }
}

class Cat extends Animal {
    @Override
    public void eat() {
        System.out.println("猫吃鱼");
    }
}

public class Test1Polymorphic {
    /*
        多态的前提:

            1. 要有(继承 \ 实现)关系
            2. 要有方法重写
            3. 要有父类引用，指向子类对象
     */
    public static void main(String[] args) {
        // 当前事物，是一只猫
        Cat c = new Cat();

        // 当前事物，是一只动物
```

```
        Animal a = new Cat();
        a.eat();

    }
}
```

3.2多态中的成员访问特点（记忆）

- 成员访问特点
  - 成员变量

    编译看父类，运行看父类
  - 成员方法

    编译看父类，运行看子类
- 代码演示

```
class Fu {
    int num = 10;

    public void method(){
        System.out.println("Fu.. method");
    }
}

class Zi extends Fu {
    int num = 20;

    public void method(){
        System.out.println("Zi.. method");
    }
}

public class Test2Polymorpic {
    /*
         多态的成员访问特点：

                成员变量：编译看左边（父类），运行看左边（父类）

                成员方法：编译看左边（父类），运行看右边（子类）
     */
    public static void main(String[] args) {
        Fu f = new Zi();
        System.out.println(f.num);
        f.method();
    }
}
```

3.3多态的好处和弊端（记忆）

- 好处

提高程序的扩展性。定义方法时候，使用父类型作为参数，在使用的时候，使用具体的子类型参与操作

- 弊端

  不能使用子类的特有成员

3.4多态中的转型（应用）

- 向上转型

  父类引用指向子类对象就是向上转型

- 向下转型

  格式：子类型 对象名 = (子类型)父类引用;

- 代码演示

```
class Fu {
    public void show(){
        System.out.println("Fu..show...");
    }
}

class Zi extends Fu {
    @Override
    public void show() {
        System.out.println("Zi..show...");
    }

    public void method(){
        System.out.println("我是子类特有的方法，method");
    }
}

public class Test3Polymorpic {
    public static void main(String[] args) {
        // 1. 向上转型 : 父类引用指向子类对象
        Fu f = new Zi();
        f.show();
        // 多态的弊端：不能调用子类特有的成员
        // f.method();

        // A: 直接创建子类对象
        // B: 向下转型

        // 2. 向下转型 : 从父类类型，转换回子类类型
        Zi z = (Zi) f;
        z.method();
    }
}
```

3.5多态中转型存在的风险和解决方案 (应用)

- 风险

如果被转的引用类型变量,对应的实际类型和目标类型不是同一种类型,那么在转换的时候就会出现ClassCastException

- 解决方案
  - 关键字

    instanceof
  - 使用格式

    变量名 instanceof 类型

    通俗的理解：判断关键字左边的变量，是否是右边的类型，返回boolean类型结果
- 代码演示

```
abstract class Animal {
    public abstract void eat();
}

class Dog extends Animal {
    public void eat() {
        System.out.println("狗吃肉");
    }

    public void watchHome(){
        System.out.println("看家");
    }
}

class Cat extends Animal {
    public void eat() {
        System.out.println("猫吃鱼");
    }
}

public class Test4Polymorpic {
    public static void main(String[] args) {
        useAnimal(new Dog());
        useAnimal(new Cat());
    }

    public static void useAnimal(Animal a){  // Animal a = new Dog();
                                             // Animal a = new Cat();
        a.eat();
        //a.watchHome();

//        Dog dog = (Dog) a;
//        dog.watchHome();  // ClassCastException  类型转换异常

        // 判断a变量记录的类型，是否是Dog
        if(a instanceof Dog){
            Dog dog = (Dog) a;
            dog.watchHome();
        }
    }
```

```
}
```

3.6黑马信息管理系统多态改进 (应用)

- 实现步骤
  1. StudentDaoFactory类中方法的返回值定义成父类类型BaseStudentDao
  2. StudentService中接收方法返回值的类型定义成父类类型BaseStudentDao
- 代码实现

  StudentDaoFactory类

```
public class StudentDaoFactory {
    public static BaseStudentDao getStudentDao(){
        return new OtherStudentDao();
    }
}
```

  StudentService类

```
public class StudentService {
    // 创建StudentDao (库管)
    // private OtherStudentDao studentDao = new OtherStudentDao();

    // 通过学生库管工厂类，获取库管对象
    private BaseStudentDao studentDao = StudentDaoFactory.getStudentDao();
}
```