JAVA复习（仅供参考）

抽象类的声明格式，并介绍抽象类和接口的区别。

```
abstract class AbstractClass {
    // 成员变量、方法的声明

    // 抽象方法的声明
    abstract void abstractMethod();

    // 普通方法的实现
    void regularMethod() {
        // 方法的实现
    }
}
```

List、Map、Set三个接口，存取元素时，各有什么特点

1. `List`：是一个有序集合，它允许存储重复的元素。在 `List` 中，每个元素都有其对应的索引，我们可以基于索引来访问和修改元素。主要的 `List` 实现类有 `ArrayList` 和 `LinkedList`。
    - `ArrayList` 是基于动态数组实现的，它在随机访问元素时性能很好，但在中间插入和删除元素时性能较差。
    - `LinkedList` 是基于双向链表实现的，它在插入和删除元素时性能很好，特别是在列表的开头或结尾，但在随机访问元素时性能较差。
2. `Set`：是一个不允许存储重复元素的集合。主要的 `Set` 实现类有 `HashSet`，`LinkedHashSet` 和 `TreeSet`。
    - `HashSet` 是基于哈希表实现的，它不保证元素的顺序，但添加，删除和查找元素的速度非常快。
    - `LinkedHashSet` 是在 `HashSet` 的基础上，添加了链表来维护元素的插入顺序。
    - `TreeSet` 是基于红黑树实现的，它保证了元素的有序性。
3. `Map`：是一个存储键值对的集合，它的每个元素都包含一个键和一个值，键不能重复，每个键对应一个值。主要的 `Map` 实现类有 `HashMap`，`LinkedHashMap` 和 `TreeMap`。
    - `HashMap` 是基于哈希表实现的，它不保证键值对的顺序。
    - `LinkedHashMap` 是在 `HashMap` 的基础上，添加了链表来维护键值对的插入顺序。
    - `TreeMap` 是基于红黑树实现的，它保证了键的有序性。

List、Set、Map接口的实现类

下面是使用 Java 中 `List`，`Set` 和 `Map` 接口的实现类的一些示例：

1. `List` 接口的实现类：

```
import java.util.List;
import java.util.ArrayList;
import java.util.LinkedList;

List<String> arrayList = new ArrayList<>();
arrayList.add("Apple");
arrayList.add("Banana");
arrayList.add("Cherry");

List<String> linkedList = new LinkedList<>();
linkedList.add("Dog");
linkedList.add("Elephant");
linkedList.add("Frog");
```

在这个例子中，我们创建了一个 `ArrayList` 和一个 `LinkedList`，并添加了一些元素。

2. `Set` 接口的实现类：

```
import java.util.Set;
import java.util.HashSet;
import java.util.LinkedHashSet;
import java.util.TreeSet;

Set<String> hashSet = new HashSet<>();
hashSet.add("India");
hashSet.add("USA");
hashSet.add("China");

Set<String> linkedHashSet = new LinkedHashSet<>();
linkedHashSet.add("Java");
linkedHashSet.add("Python");
linkedHashSet.add("C++");

Set<String> treeSet = new TreeSet<>();
treeSet.add("Lion");
treeSet.add("Tiger");
treeSet.add("Elephant");
```

在这个例子中，我们创建了一个 `HashSet`，一个 `LinkedHashSet` 和一个 `TreeSet`，并添加了一些元素。

3. `Map` 接口的实现类：

```
import java.util.Map;
import java.util.HashMap;
import java.util.LinkedHashMap;
import java.util.TreeMap;

Map<String, Integer> hashMap = new HashMap<>();
hashMap.put("Alice", 25);
hashMap.put("Bob", 30);
hashMap.put("Charlie", 35);

Map<String, Integer> linkedHashMap = new LinkedHashMap<>();
linkedHashMap.put("Apple", 100);
```

```
linkedHashMap.put("Banana", 200);
linkedHashMap.put("Cherry", 300);

Map<String, Integer> treeMap = new TreeMap<>();
treeMap.put("Cat", 1);
treeMap.put("Dog", 2);
treeMap.put("Elephant", 3);
```

Iterator接口中的主要方法

Iterator接口是Java集合框架中的一个接口，它定义了一组用于遍历集合元素的方法。以下是Iterator接口中的主要方法：

1. `boolean hasNext()`：判断集合中是否还有下一个元素可供遍历。如果有下一个元素，则返回true；否则返回false。
2. `E next()`：返回集合中的下一个元素，并将迭代器的指针向后移动一位。如果没有下一个元素可供遍历，则会抛出NoSuchElementException异常。
3. `void remove()`：从集合中移除迭代器最后一次返回的元素。如果在调用next()之前没有调用过remove()，或者在上一次调用next()之后已经调用了remove()，则会抛出IllegalStateException异常。

这些方法使得我们可以使用Iterator来遍历集合元素，而不需要了解具体的集合实现细节。通过调用hasNext()方法判断是否还有下一个元素可供遍历，然后使用next()方法获取下一个元素，并使用remove()方法在需要时移除元素。

```
import java.util.ArrayList;
import java.util.Iterator;

public class IteratorExample {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("Apple");
        list.add("Banana");
        list.add("Orange");

        Iterator<String> iterator = list.iterator();
        while (iterator.hasNext()) {
            String element = iterator.next();
            System.out.println(element);
        }
    }
}
```

在上述示例中，我们首先创建了一个ArrayList集合，并添加了几个元素。然后使用iterator()方法获取该集合的迭代器对象。接下来，在while循环中使用hasNext()方法判断是否还有下一个元素可供遍历，然后使用next()方法获取下一个元素并打印出来。

请注意，使用Iterator遍历集合时，如果在遍历过程中修改了集合的结构（比如添加或删除元素），可能会导致"ConcurrentModificationException"异常。因此，在遍历过程中应该避免直接使用集合的add()、remove()等方法，而是使用Iterator的remove()方法来安全地移除元素。

面向对象的基本特征

面向对象编程（Object-Oriented Programming，OOP）是一种编程范式，其基本特征主要包括以下几点：

1. **封装（Encapsulation）**：封装是指将对象的状态（属性）和行为（方法）打包在一起，并隐藏对象内部的实现细节。封装可以防止外部代码随意访问对象内部的状态，而是通过提供的方法来进行操作。
2. **继承（Inheritance）**：继承是指一个类（子类）可以继承另一个类（父类）的属性和方法。子类可以复用父类的代码，提高代码的重用性，并允许子类添加或重写父类的行为。
3. **多态（Polymorphism）**：多态是指子类对象可以替代父类对象使用，同一消息可以被多个对象以不同的方式解释。这样可以增强程序的扩展性和灵活性。
4. **抽象（Abstraction）**：抽象是指只展示对象的重要特性，而隐藏不必要的细节。抽象可以通过接口和抽象类来实现。接口定义了一种契约，规定了实现接口的类应该提供哪些服务。抽象类是不能被实例化的类，它通常包含一些未完全定义的抽象方法，供子类实现。

结构化程序设计的3大基本结构

结构化程序设计是一种编程范式，它强调使用三种基本结构来构建程序，以实现清晰、可读性强、易于维护和调试的代码。这三大基本结构是：

1. **顺序结构（Sequence）**：顺序结构是程序中最简单的结构，它按照代码的书写顺序依次执行语句。代码按照从上到下的顺序逐行执行，没有分支或循环。顺序结构是程序的默认结构，它用于按照特定的顺序执行一系列操作或语句。
2. **选择结构（Selection）**：选择结构用于根据条件的真假来选择执行不同的代码块。常用的选择结构是if语句，它根据条件表达式的结果选择性地执行某个代码块。另外，还有switch语句可以根据不同的取值选择性地执行不同的代码块。
3. **循环结构（Iteration）**：循环结构用于重复执行一段代码块，直到满足特定条件为止。循环结构允许程序多次执行相同或类似的操作，以便处理大量的数据或重复的任务。常用的循环结构有for循环、while循环和do-while循环，它们根据不同的条件判断来决定是否继续循环。

这三大基本结构可以组合使用，构建出复杂的程序逻辑。通过合理地使用顺序、选择和循环结构，我们可以编写出结构良好、逻辑清晰的程序。这种结构化的编程方式有助于提高代码的可读性、可维护性和可扩展性，使程序更易于理解和调试。

Java提供的几种循环

Java提供了多种循环结构，包括以下几种常用的循环：

1. for循环：for循环用于重复执行一段代码，具有明确的循环次数。它由初始化表达式、循环条件和循环迭代部分组成。

```
for (int i = 0; i < 5; i++) {
    System.out.println("Iteration " + i);
}
```

2. while循环：while循环用于在满足条件的情况下重复执行一段代码。它在循环开始之前先判断条件是否为真，如果为真则执行循环体，然后再次判断条件。

```
int i = 0;
while (i < 5) {
    System.out.println("Iteration " + i);
    i++;
}
```

3. do-while循环：do-while循环类似于while循环，但是它先执行循环体，然后再判断条件是否为真。因此，无论条件是否为真，循环体至少会执行一次。

```
int i = 0;
do {
    System.out.println("Iteration " + i);
    i++;
} while (i < 5);
```

4. 增强的for循环（foreach循环）：增强的for循环用于遍历数组或集合中的元素，它提供了一种简化的语法，使得遍历更加方便。

```
int[] numbers = {1, 2, 3, 4, 5};
for (int number : numbers) {
    System.out.println(number);
}
```

以上是Java中常用的几种循环结构。根据具体的需求和循环条件，选择合适的循环结构来完成任务。

运行时多态的优点

1. **灵活性和扩展性**：运行时多态使得代码可以以一种抽象的方式编写，而不依赖于具体的实现细节。这样可以提高代码的灵活性，使得程序更容易扩展和修改。通过使用多态，可以在不修改现有代码的情况下，添加新的子类实现或修改现有的实现，从而实现代码的可维护性和可扩展性。
2. **代码重用**：多态性允许通过继承和接口实现代码的重用。通过定义通用的抽象类或接口，并让多个具体的子类实现相同的接口或继承相同的抽象类，可以减少代码的重复编写。这样可以提高代码的复用性和维护性。
3. **可替代性和可扩展性**：多态性使得我们可以通过定义通用的接口或抽象类来编写代码，而不依赖于具体的实现类。这样可以使得程序的各个模块之间解耦，提高模块的可替代性和可扩展性。如果需要更换或新增一种实现方式，只需要提供相应的具体实现类，并在运行时将其传递给使用方即可。
4. **实现多态的方式**：多态可以通过继承和接口实现。继承实现的多态性可以通过父类引用指向子类对象来实现，而接口实现的多态性可以通过实现接口并使用接口引用来实现。这种多态的实现方式使得代码更加灵活，可以根据具体的需求选择适合的实现方式。

运行时多态性是面向对象编程的重要特性，它提供了代码的灵活性、扩展性、重用性和可替代性。通过合理使用多态，可以编写出高度可维护、可扩展和可复用的代码，提高软件开发的效率和质量。

Arrays类中提供的数组排序方法

1. `sort()`：对数组进行升序排序。对于基本数据类型的数组，使用快速排序算法；对于对象数组，使用对象的自然排序或指定的比较器进行排序。

```
int[] numbers = {5, 2, 8, 1, 4};
Arrays.sort(numbers);
System.out.println(Arrays.toString(numbers)); // 输出：[1, 2, 4, 5, 8]
```

2. `sort(T[] a, Comparator<? super T> c)`：对对象数组根据指定的比较器进行排序。

```
String[] names = {"John", "Alice", "Bob", "David"};
Arrays.sort(names, (s1, s2) -> s1.compareToIgnoreCase(s2));
System.out.println(Arrays.toString(names)); // 输出：[Alice, Bob, David, John]
```

3. `parallelSort()`：对数组进行并行排序。与`sort()`方法类似，但使用并行算法以提高排序性能。仅适用于基本数据类型的数组。

```
int[] numbers = {5, 2, 8, 1, 4};
Arrays.parallelSort(numbers);
System.out.println(Arrays.toString(numbers)); // 输出：[1, 2, 4, 5, 8]
```

4. `parallelSort(T[] a, Comparator<? super T> c)`：对对象数组根据指定的比较器进行并行排序。

```
String[] names = {"John", "Alice", "Bob", "David"};
Arrays.parallelSort(names, (s1, s2) -> s1.compareToIgnoreCase(s2));
System.out.println(Arrays.toString(names)); // 输出：[Alice, Bob, David, John]
```

这些方法可以方便地对数组进行排序操作，无需手动实现排序算法。注意，这些排序方法会直接修改原始数组，而不是返回一个新的排序后的数组。如果需要保留原始数组，可以在排序之前创建一个副本进行操作。另外，排序的对象需要实现`Comparable`接口或使用指定的比较器进行比较。

标识符的命名规则

在Java中，标识符（Identifier）是用来命名变量、方法、类、包等程序实体的名称。命名规则为：

1. 由字母、数字、下划线（_）和美元符号（$）组成。
2. 必须以字母、下划线或美元符号开头。
3. 标识符区分大小写，即大小写字母是不同的。
4. 不能使用Java的保留关键字作为标识符，如`public`、`class`等。
5. 标识符应具有描述性和可读性，以便于代码理解和维护。
6. 标识符不能包含空格或特殊字符，如@、#、%等。
7. Java建议使用驼峰命名规则（CamelCase）命名标识符，即首字母小写，后续每个单词的首字母大写。

一些符合Java标识符命名规则的标识符：

```
int age;
String firstName;
double averageSalary;
myVariable;
total_count;
```

一些不符合Java标识符命名规则的标识符：

```
3numbers; // 数字开头
hello world; // 包含空格
public; // 保留关键字
```

异常抛出

在Java中，可以使用`throw`关键字抛出异常。抛出异常的过程可以用以下步骤表示：

1. 创建异常对象：首先，需要创建一个表示特定异常情况的异常对象。可以使用Java提供的内置异常类，如`Exception`、`RuntimeException`等，或者自定义异常类。
2. 抛出异常：使用`throw`关键字后跟要抛出的异常对象来抛出异常。抛出异常后，程序会立即停止当前方法的执行，并将异常传递给调用者。
3. 处理异常：在抛出异常的方法外部，可以使用`try-catch`语句块来捕获和处理异常。通过捕获异常，程序可以继续执行其他代码而不会终止。

以下是一个示例代码，演示如何抛出和捕获异常：

```
public class ExceptionExample {
    public static void main(String[] args) {
        try {
            int result = divide(10, 0);
            System.out.println("Result: " + result);
        } catch (ArithmeticException ex) {
            System.out.println("Error: " + ex.getMessage());
        }
    }

    public static int divide(int dividend, int divisor) throws
ArithmeticException {
        if (divisor == 0) {
            throw new ArithmeticException("Division by zero is not allowed.");
        }
        return dividend / divisor;
    }
}
```

在上述示例中，`divide()`方法用于执行除法操作。如果除数为0，则会抛出`ArithmeticException`异常，并传递错误消息。在`main()`方法中，通过调用`divide()`方法来执行除法操作，并使用`try-catch`语句块捕获异常。如果捕获到异常，会输出错误消息；否则，会打印除法的结果。

注意，`throws`关键字用于方法签名中，表示该方法可能抛出指定的异常类型。在上述示例中，`divide()`方法声明了可能抛出`ArithmeticException`异常。在方法内部抛出异常时，可以使用`throw`关键字将异常抛出。

If else语句的运用，如分段函数

使用if-else语句可以实现分段函数的逻辑。分段函数是指根据不同的输入值，在不同的范围内使用不同的计算公式或逻辑来得到输出结果。下面是一个使用if-else语句实现分段函数的示例代码：

```
public class PiecewiseFunction {
```

```
    public static void main(String[] args) {
        double x = 4.5;
        double result;

        if (x <= 0) {
            result = 2 * x;
        } else if (x > 0 && x < 5) {
            result = x * x + 3;
        } else {
            result = 5 * x - 1;
        }

        System.out.println("Result: " + result);
    }
}
```

在上述示例中，我们根据输入值x的不同情况，采用不同的计算公式来计算输出结果result。if-else语句中的条件表达式用于判断x所属的范围，并执行相应的计算逻辑。

- 如果x小于等于0，执行`result = 2 * x`;
- 如果x大于0且小于5，执行`result = x * x + 3`;
- 如果x大于等于5，执行`result = 5 * x - 1`。

最后，通过输出语句打印结果result。

通过使用if-else语句实现分段函数，可以根据具体的输入值执行不同的逻辑，从而得到不同的输出结果。这种方式可以方便地处理复杂的条件逻辑，使代码更加清晰和可读。

数组的定义，输入，计算，如将考试成绩输入一维数组，并求平均成绩及不及格学生的人数

要实现将考试成绩输入一维数组，并求平均成绩及不及格学生的人数，可以按照以下步骤进行：

1. 定义数组：声明一个一维数组来存储考试成绩。数组的长度可以根据实际需求进行定义，比如假设有10个学生的成绩，可以定义一个长度为10的数组。

```
int[] scores = new int[10];
```

2. 输入成绩：使用循环结构（如for循环）来逐个输入学生的成绩，并将其存储在数组中。

```
import java.util.Scanner;

Scanner scanner = new Scanner(System.in);

for (int i = 0; i < scores.length; i++) {
    System.out.print("请输入第 " + (i + 1) + " 个学生的成绩：");
    scores[i] = scanner.nextInt();
}
```

3. 计算平均成绩和不及格人数：使用循环遍历数组，累加成绩求和，并统计不及格成绩的人数。

```
int sum = 0;
int count = 0;
```

```
for (int i = 0; i < scores.length; i++) {
    sum += scores[i];

    if (scores[i] < 60) {
        count++;
    }
}

double average = (double) sum / scores.length;

System.out.println("平均成绩：" + average);
System.out.println("不及格人数：" + count);
```

在上述代码中，我们使用循环遍历数组`scores`，将每个学生的成绩进行累加，并通过计算平均值得到平均成绩。同时，使用一个计数器`count`统计不及格成绩的人数。

最后，通过输出语句打印平均成绩和不及格人数的结果。

请注意，在实际应用中，可能需要对输入进行验证和错误处理，确保输入的成绩符合要求。此处的示例代码仅为演示目的，并未包含输入验证。

类的定义，在类中定义属性和方法

在Java中，类是一种面向对象的编程概念，用于表示具有共同特征和行为的对象集合。类定义了对象的属性（成员变量）和方法（成员函数）。下面是一个示例，展示了如何定义一个简单的Java类，并在类中定义属性和方法：

```
public class MyClass {
    // 成员变量
    private int number;
    private String name;

    // 构造方法
    public MyClass(int number, String name) {
        this.number = number;
        this.name = name;
    }

    // 成员方法
    public void printInfo() {
        System.out.println("Number: " + number);
        System.out.println("Name: " + name);
    }

    public int getNumber() {
        return number;
    }

    public void setNumber(int number) {
        this.number = number;
    }

    public String getName() {
        return name;
```

```
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

在上述示例中，我们定义了一个名为`MyClass`的类。该类具有两个私有成员变量`number`和`name`，用于存储对象的数据。我们还定义了一个构造方法，用于初始化对象的属性。

该类还包含了一些成员方法，其中`printInfo()`方法用于打印对象的属性信息，`getNumber()`和`getName()`方法用于获取属性值，`setNumber()`和`setName()`方法用于设置属性值。

注意，成员变量被声明为私有（private），这意味着它们只能通过公共的访问方法（getter和setter）来访问和修改。这是一种封装的概念，通过封装可以控制对对象的访问和操作。

通过定义类的属性和方法，我们可以创建类的实例（对象），并通过对象调用方法来执行相应的操作。例如：

```
MyClass obj = new MyClass(42, "John Doe");
obj.printInfo();

obj.setNumber(100);
obj.setName("Jane Smith");

int number = obj.getNumber();
String name = obj.getName();
```

上述代码创建了一个`MyClass`对象，并通过构造方法初始化属性。然后调用`printInfo()`方法打印对象的属性信息。接下来，使用`setNumber()`和`setName()`方法修改属性的值，并使用`getNumber()`和`getName()`方法获取属性值。

这样，通过定义类和使用对象，我们可以组织和操作数据，实现更加模块化和可维护的代码结构。

商店打折售货

1. 定义商品类：首先，定义一个表示商品的类，该类包含商品的属性（如名称、价格等）和方法（如计算折扣后价格）。

```
public class Product {
    private String name;
    private double price;

    public Product(String name, double price) {
        this.name = name;
        this.price = price;
    }

    public String getName() {
        return name;
    }

    public double getPrice() {
        return price;
```

```
    }

    public double getDiscountedPrice(double discountPercentage) {
        double discountAmount = price * (discountPercentage / 100);
        return price - discountAmount;
    }
}
```

在上述示例中，定义了一个`Product`类，具有名称和价格属性，并提供了构造方法和获取属性值的方法。还定义了一个`getDiscountedPrice()`方法，根据给定的折扣百分比计算折扣后的价格。

2. 创建商品对象：使用定义的商品类，创建具体的商品对象，并设置初始价格。

```
Product laptop = new Product("Laptop", 1000.0);
Product smartphone = new Product("Smartphone", 500.0);
```

3. 应用折扣并计算价格：根据商店的折扣策略，使用商品对象的`getDiscountedPrice()`方法计算折扣后的价格。

```
double discountPercentage = 10.0; // 假设商店打九折

double laptopDiscountedPrice = laptop.getDiscountedPrice(discountPercentage);
double smartphoneDiscountedPrice =
smartphone.getDiscountedPrice(discountPercentage);
```

在上述示例中，假设商店打九折，计算了笔记本电脑和智能手机的折扣价格。

4. 输出结果：通过输出语句打印折扣后的价格。

```
System.out.println("Laptop Discounted Price: $" + laptopDiscountedPrice);
System.out.println("Smartphone Discounted Price: $" +
smartphoneDiscountedPrice);
```

最后，通过输出语句将折扣后的价格打印出来。

以上是一个简单的示例，展示了如何实现JAVA商店的打折售货。根据实际需求，可以进一步扩展和调整代码，以适应更复杂的商店折扣策略和商品信息管理。

类的继承和实现关键字，定义格式

在 Java 中，你可以创建类、抽象类以及接口。以下是各自的简单实现：

1. 类的实现：

```
public class MyClass {
    private int value;

    public MyClass(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }
```

```
    public void setValue(int value) {
        this.value = value;
    }
}
```

在这个例子中，`MyClass` 是一个简单的类，包含一个私有字段 `value`，以及用于获取和设置 `value` 的公共方法。

2. 抽象类的实现：

```
public abstract class MyAbstractClass {
    public abstract void myAbstractMethod();
}
```

在这个例子中，`MyAbstractClass` 是一个抽象类，它有一个没有具体实现的抽象方法 `myAbstractMethod`。任何继承 `MyAbstractClass` 的类都需要提供 `myAbstractMethod` 的具体实现。

3. 接口的实现：

```
public interface MyInterface {
    void myInterfaceMethod();
}
```

在这个例子中，`MyInterface` 是一个接口，它定义了一个名为 `myInterfaceMethod` 的方法。任何实现 `MyInterface` 的类都需要提供 `myInterfaceMethod` 的具体实现。

这些都是非常简单的实现，实际上类、抽象类和接口都可以更复杂，包含更多的方法和字段，也可以包含内部类、内部接口等。如果你想在 MyClass 中使用接口或抽象类，你需要让 MyClass 实现接口或者继承抽象类，并实现所有的抽象方法。以下是一个例子：

1. 实现接口：

```
public interface MyInterface {
    void myInterfaceMethod();
}

public class MyClass implements MyInterface {
    public void myInterfaceMethod() {
        // 这里是你的具体实现
    }
}
```

在这个例子中，`MyClass` 实现了 `MyInterface` 接口，并提供了 `myInterfaceMethod` 方法的实现。

2. 继承抽象类：

```
public abstract class MyAbstractClass {
    public abstract void myAbstractMethod();
}

public class MyClass extends MyAbstractClass {
    public void myAbstractMethod() {
        // 这里是你的具体实现
    }
}
```

在这个例子中，`MyClass` 继承了 `MyAbstractClass` 抽象类，并提供了 `myAbstractMethod` 方法的实现。

注意：一个类只能继承一个抽象类，但可以实现多个接口。

实验题1

```
/**
*   @author NingYin
*/
package First;

import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        int x = input.nextInt();
        int y=0;
        if(x>20){
            y = x + 5;
        }else if(x<20 && x>=0){
            y = x - x * x ;
        }else if(x<0){
            y = x - 8;
        }else{
            System.out.println("Input invalid.");
        }
        System.out.println(y);
    }
}
```

实验题2

```
/**
*   @author NingYin
*/
package Second;

import java.util.Scanner;

public class Main {
    public static void main(String[] args) throws Exception {
```

```
        int num = 60;
        Scanner input = new Scanner(System.in);
        float[] score = new float[num];
        float totalscore = 0;
        float averagescore = 0;
        int badstudent = 0;
        for(int i=0;i<num;i++){
            score[i] = input.nextFloat();
            if(score[i] >=0 && score[i] <=100){
                totalscore += score[i];
                if(score[i] < 60){
                    badstudent++;
                }
            }else{
                throw new Exception("分数错误");
            }
        }
        averagescore = totalscore / num;
        System.out.println("平均成绩为：" + averagescore);
        System.out.println("低于60分的人数为：" + badstudent);
    }
}
```

实验题3

```
/**
*   @author NingYin
*/
package Third;

import java.util.Scanner;

public class Main {
    public static void main(String[] args) throws Exception{
        boolean sex = false;
        System.out.println("输入1为男性，输入0为女性");
        Scanner input = new Scanner(System.in);
        int sexis = input.nextInt();
        if(sexis == 1){
            sex = true;
        }else if (sexis == 0){
            sex = false;
        }else{
            throw new Exception("输入有误，请重试");
        }
        Teacher teacher = new Teacher();
        teacher.Introduce(sex);
    }
}
```

```
/**
*   @author NingYin
```

```java
*/
package Third;

public class Teacher {
    private float height = 0;
    private String college = "无锡学院";
    private boolean sex = false;
    void getters(){}
    void setters(){}
    public void Introduce(boolean sex){
        //this.height = 1.65F; //修改私有变量
        this.sex = sex;
        if(this.sex == true){
            System.out.println("他是男老师");
        }else{
            System.out.println("她是女老师");
        }
    }
}
```