1. **Классы как типы данных.**

Реализовать класс Vector3D, представляющий вектор в 3-хмерном пространстве. У класса должны быть следующие методы:

// Получение координат

double getX();

double getY();

double getZ();

// Операции

double scalar(Vector3D v); // скалярное произведение векторов

double len(); // длина вектора

Vector3D multiply(double factor); // умножение на число

Vector3D add(Vector3D v); // сложение векторов

Vector3D sub(Vector3D v); // вычитание векторов

Vector3D vecX(Vector3D v); // векторное произведение векторов

boolean pcollin(Vector3D v); // проверка на коллинеарность

В классе Vector3D переопределите метод equals() для сравнения значения полей.

Класс вектора должен быть покрыт модульными тестами, сделанными на основе библиотеки JUnit.

Затем следует реализовать класс Segment, представляющий отрезок в 3-хмерном пространстве. У класса должны быть следующие методы:

double len();// длина отрезка

double distanceTo(Vector3D point); ();// расстояние от точки до отрезка

В отдельном классе SegmentTest опишите метод main(...), в котором вводим с консоли координаты отрезка, координаты точки, на экран выводится длина отрезка, расстояние от точки до отрезка.

**Сравнение вещественных чисел**

f1 = 1.0999999999999999

f2 = 1.1

f1 и f2 не равны!

Подобные ошибки связаны с тем, как числа представлены в двоичном виде в памяти компьютера.

Итак, правило №1 при сравнении вещественных чисел:

**никогда не используй == при сравнении чисел с плавающей точкой.**

**public** **class** Main {

**public** **static** **void** main(String[] args) {

**final** **double** threshold = 0.0001;

       //прибавляем к нулю 0.1 одиннадцать раз подряд

**double** f1 = .0;

**for** (**int** i = 1; i <= 11; i++) {

           f1 += .1;

       }

       //Умножаем 0.1 на 11

**double** f2 = .1 \* 11;

       System.out.println("f1 = " + f1);

       System.out.println("f2 = " + f2);

**if** (Math.abs(f1 -f2) < threshold)

           System.out.println("f1и f2 равны");

**else**

           System.out.println("f1и f2 не равны");

   }

}

Меняем способ сравнения чисел.

Есть специальное «пороговое» число — 0.0001, одна десятитысячная. Оно может быть и другим. Это зависит от того, насколько точное сравнение тебе нужно в конкретном случае. Можно сделать его и больше, и меньше.

 С помощью метода Math.abs()получаем модуль числа.

Вычитаем второе число из первого, и если полученный результат, независимо от знака, будет меньше того порога, который мы установили, значит наши числа равны.

**Во всяком случае, они равны до той степени точности, которую мы установили с помощью нашего «порогового числа»**, то есть как минимум они равны вплоть до одной десятитысячной. Такой способ сравнения избавит от неожиданного поведения, которое мы увидели в случае с ==.

Переопределение equals() и hashCode()

*Переопределение метода* (method overriding) — это приём при котором поведение родительского класса или интерфейса переписывается (переопределяется) в подклассе. В Java у каждого объекта есть методы equals() и hashCode() и для правильной работы они должны быть переопределены.

Чтобы понять, как работает переопределение equals() и hashCode(), изучим их реализацию в базовых классах Java. Ниже приведён метод equals() класса Object. Метод проверяет, совпадает ли текущий экземпляр с переданным объектом obj.

public boolean equals(Object obj) {

        return (this == obj);

}

Теперь посмотрим на метод hashCode() в классе Object.

@HotSpotIntrinsicCandidate

public native int hashCode();

Это native — метод, который написан на другом языке, таком как Си, и он возвращает некоторый числовой код, связанный с адресом памяти объекта. (Если вы не пишете код JDK, то не важно точно знать, как работает этот метод.) 

Если методы equals() и hashCode() не переопределены, вместо них будут вызваны методы класса Object, описанные выше. В этом случае методы не выполняют реальной цели equals() и hashCode(), которая состоит в том, чтобы проверить, имеют ли объекты одинаковые состояния.

Как правило, при переопределении equals() также переопределяется hashCode().

Сравнение объектов с equals()

Метод equals() используется для сравнения объектов. Чтобы определить одинаковые объекты или нет, equals() сравнивает значения полей объектов:

public class EqualsAndHashCodeExample {

  public static void main(String... args){

    System.out.println(new Simpson("Homer", 35, 120)

      .equals(new Simpson("Homer",35,120)));

    System.out.println(new Simpson("Bart", 10, 120)

      .equals(new Simpson("El Barto", 10, 45)));

    System.out.println(new Simpson("Lisa", 54, 60)

      .equals(new Object()));

  }

  static class Simpson {

    private String name;

    private int age;

    private int weight;

    public Simpson(String name, int age, int weight) {

        this.name = name;

        this.age = age;

        this.weight = weight;

    }

    @Override

    public boolean equals(Object o) {

*// 1*

        if (this == o) {

            return true;

        }

*// 2*

        if (o == null || getClass() != o.getClass()) {

            return false;

        }

*// 3*

        Simpson simpson = (Simpson) o;

        return age == simpson.age &&

               weight == simpson.weight &&

               name.equals(simpson.name);

    }

  }

}

Посмотрим на метод equals(). Первое сравнение сравнивает текущий экземпляр объекта this с переданным объектом o. Если это один и тот же объект, то equals() вернёт true.

Во втором сравнении проверяется, является ли переданный объект null и какой у него тип. Если переданный объект другого типа, то объекты не равны.

Наконец, equals() сравнивает поля объектов. Если два объекта имеют одинаковые значения полей, то объекты совпадают.

Анализ вариантов сравнения объектов

Теперь давайте посмотрим на варианты сравнений объектов в методе main(). Сначала мы сравниваем два объекта Simpson:

System.out.println(

    new Simpson("Homer", 35, 120).equals(

    new Simpson("Homer", 35, 120)));

У полей этих объектов одинаковые значения, поэтому результат будет true.

Затем снова сравниваем два объекта Simpson:

System.out.println(

    new Simpson("Bart", 10, 45).equals(

    new Simpson("El Barto", 10, 45)));

Объекты здесь похожи, но значения имён разные: *Bart* и *El Barto*. Поэтому результат будет false.

Наконец, давайте сравним объект Simpson и экземпляр класса [Object](https://docs.oracle.com/javase/8/docs/api/java/util/Objects.html" \t "_blank):

System.out.println(

    new Simpson("Lisa", 54, 60).equals(

    new Object()));

В этом случае результат будет false, так как типы объектов отличаются.

equals() в сравнении с ==

На первый взгляд кажется, что оператор == и метод equals() делают одно и то же, но, на самом деле, они работают по-разному. Оператор == сравнивает, указывают ли две ссылки на один и тот же объект. Например:

Simpson homer  = new Simpson("Homer", 35, 120);

Simpson homer2 = new Simpson("Homer", 35, 120);

System.out.println(homer == homer2);

Мы создали два разных экземпляра Simpson с помощью оператора new. Поэтому переменные homer и homer2 будут указывать на разные объекты в [куче](https://www.baeldung.com/java-stack-heap). Таким образом, в результате получим false.

Во следующем примере используем переопределенный метод equals():

System.out.println(homer.equals(homer2));

В этом случае будут сравниваться поля. Поскольку значения полей у обоих объектов Simpson одинаковые, результат будет true.

**Тестирование консольного калькулятора Java с помощью JUnit**

Для тестирования возьмём консольный калькулятор Java  для которого и напишем  **JUnit Test**.

**class Calc** реализовывает метод, который будет выполнять действия над числами в зависимости от знака операции.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | class Calc  {  public static int metodCalc(int a1, String op, int a2)  {  int res=0;  switch(op)  {  case "+": res= a1 + a2;  break;  case "-": res= a1 - a2;  break;  case "\*": res= a1 \* a2;  break;  case "/": res= a1 / a2;  break;  }  return res;  }  } |

 В нашем калькуляторе мы будем проверять модуль который выполняет операции ( + , – , \* , / )

Проверим: правильно наш метод выполняет операцию прибавления цифр.

Для этого напишем следующие строки кода:

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import static org.junit.Assert.\*;    import org.junit.Test;      public class Test\_1  {  @org.junit.Test  public void testPlus()  {  int res= Calc.metodCalc(15, "+", 5);  assertEquals(20, res);  }  } |

В переменную **res** мы записываем результат, который нам выдаст метод ***metodCalc***из класса ***Calc****.*С помощью метода ***assertEquals***мы сравниваем результат, который посчитал метод **Calc.*metodCalc*** с ожидаемым результатом, который мы написали сами.

Кратко о том, как легко подключить JUnit в IntelliJ IDEA

Однако IDEA предлагает гораздо больше. Если у вас еще нет тестового класса и вы хотите создать его для любого из исходных классов, см. инструкции ниже.

Вы можете использовать действие **создать тестовое** намерение, нажав Alt + Enter , стоя на имени вашего класса внутри редактора, или с помощью Ctrl + Shift + T сочетание клавиш.

Появится диалоговое окно, в котором вы выбираете, какую платформу тестирования использовать, и впервые нажимаете кнопку **Fix** , чтобы добавить необходимые библиотечные банки в зависимости модуля. Вы также можете выбрать методы для создания тестовых заглушек.



