1. 前言

这门功课如果你前面在刚接触Java的时候，或者学习其他语言的时候理解了面向对象编程的意思，那么学习这门功课你就会觉得很简单。当然如果你至今没有理解面向对象编程，这又是一次理解面向对象编程的好机会。无论你对面向对象编程的理解如何，这门功课都是一次对面向对象编程的练习和实践。希望在学习完这门功课后，你能在今后自己的代码中能够做到面向对象编程，面向对象编程并不是什么繁文缛节，而是能够帮助你更好地理解这个世界，更好地重构这个世界，写出更加简洁使人容易理解的代码出来。
这门功课使用的语言依旧是Java语言，Java语言是一种高级面向对象编程语言，它能充分体现出面向对象编程这一点，同时Java语言很热门，使用面很广，所以你学习完Java对今后的工作以及学习其他的语言都是有很大帮助的，希望大家能好好学习这门功课，因为这门功课简单又重要。
前几节课是在复习Java语言，这一部分也会在这里以讲解和练习的形式帮助大家进行复习。

2. Arrays（数组）

数组可能是大家第一次接触到的数据结构，这种数据结构的意义是什么呢？我们之前学习过用变量存储数据了，比如我们用一个变量num储存了一个数字100，但是如果我们需要储存100个数字呢，如果我们使用变量去存储，这里我们就需要100个变量去存储这些数字，而且这里有100个变量我们也不方便调用，它们既然都是数字，那我们就可以用一种数据结构去存储这样的一系列数据，这些数据是同样的类型。所以现在我们使用一个数组，这个数组可以存储这个100个数字，而且我们在调用的时候直接调用这个数组也很方便（即我们很容易计算这里100个数字的和或者平均值等）。这就是数组的意义。

2.1 声明（Declaration）

首先需要声明数组变量，声明的语法主要有两种。

```
dataType[] arrayRefVar;// 首选的方法
dataType arrayRefVar[];// 效果相同，但不是首选方法
```

下面的方法来自 C/C++ 语言 ，在Java中采用是为了让 C/C++ 程序员能够快速理解java语言。
因此例子如下。

```
double[] myList;
double myList[];
```

2.2 创建（Creation）

数组变量的创建使用new操作符来进行。

```
arrayRefVar = new dataType[arraySize];
```

这里使用new操作符创建了数组后又将新创建的数组的引用赋值给变量arrayRefVar。这里注意是引用，即它的内存地址，后面我们会详细叙述这一点。
数组变量的声明和创建可以用一条语句完成，如下。

```
dataType[] arrayRefVar = new dataType[arraySize];
```

所以我们的示例如下。

```
int[] array1;
array1 = new int[10];

int[] array2 = new int[10]
```

当数组被创建后，里面有默认值而非直接都是空。
对于基础数据类型（Primitive Data Types）：
byte：默认值为 0
short：默认值为 0
int：默认值为 0
long：默认值为 0L
float：默认值为 0.0f
double：默认值为 0.0d
char：默认值为 \u0000（即空字符）
boolean：默认值为 false
其余的默认值为null
默认值的存在方便了我们初始化（Initialization），初始化（Initialization）通常是指给变量或数据结构赋予初始值的过程，但也可以包含声明（Declaration）、创建（Creation）和赋值（Assignment）这几个步骤。通过这个定义我们也能发现其方便了我们赋值的过程。否则比如我们要实时计算数组内所有值的和，如果没有默认值的话就会出现报错（null出现在了计算中）。

我们也可以用下面这种方式创建数组。

```
dataType[] arrayRefVar = {value0, value1, ..., valuek};
```

示例如下。

```
double[] myList = {1.9, 2.9, 3.4, 3.5};
```

它等价于下列代码。

```
 double[] myList = new double[4];
 myList[0] = 1.9;
 myList[1] = 2.9;
 myList[2] = 3.4;
 myList[3] = 3.5;
```

当然下列代码也是和上面等价的。

```
double[] myList = new double[]{1.9, 2.9, 3.4, 3.5};
```

2.3 索引（Index）

我们通过索引访问（access）数组中的特定元素。访问的意思是允许我们按需读取或修改这些元素。
首先索引是从0开始的，这意味着数组的第一个元素的索引是0，最后一个元素的索引是数组的长度-1。
数组的长度在创建的时候就被固定下来了，无法更改。如果你想获取一个数组的大小你可以使用下面的方式。

```
arrayRefVar.length
```

下面的图片描绘了数组myList。这里 myList 数组里有10个double类型的元素，它的索引是从0-9。

因此我们这里输入代码。

```
System.out.println(myList.length);// 输出的结果是10
System.out.println(myList[0]);//输出的结果是5.6
myList[0] = 1.2;
myList[1] = 2.2;
myList[2]= myList[0] + myList[1];
System.out.println(myList[2]);//输出的结果是3.4
```

2.4 实践

下面的示例代码中展示了关于数组的多种使用。

```
public class TestArray {
   public static void main(String[] args) {
      double[] myList = new double[10];
      // 使用输入的值初始化数组
      Scanner input = new Scanner(System.in);
      System.out.print("Enter " + myList.length + " values: ");
      for (int i = 0; i < myList.length; i++) {
        myList[i] = input.nextDouble();
      }
      // 打印所有数组元素
      for (int i = 0; i < myList.length; i++) {
         System.out.println(myList[i] + " ");
      }
      // 计算所有元素的总和
      double total = 0;
      for (int i = 0; i < myList.length; i++) {
         total += myList[i];
      }
      System.out.println("Total is " + total);
      // 查找最大元素
      double max = myList[0];
      for (int i = 1; i < myList.length; i++) {
         if (myList[i] > max) {
            max = myList[i];
         }
      }
      System.out.println("Max is " + max);
   }
}
```

2.4.1 移动数组里的值

如果我们想将数组里的值进行移动。
向左移动，第一个元素移动至最后一个的代码如下。

```
int temp = myList[0];
for (int i = 1; i < myList.length; i++) {
    myList[i - 1] = myList[i];
}
myList[myList.length - 1] = temp;
```

这里需要注意索引不能超过索引的上限，如下面代码中就会出现ArrayIndexOutOfBoundsException的报错。

```
for (int i = 0; i < myList.length; i++) {
    myList[i] = myList[i + 1];
}
```

这一点在使用数组的时候需要注意。

向右移动需要注意的是不能从左向右进行，否则所有的值都会变成一样的，因为我们需要移动的值被这一操作覆盖了。

```
int temp = myList[myList.length - 1];
for (int i = myList.length - 1; i > 0; i--) {
    myList[i] = myList[i - 1];
}
myList[0] = temp;
```

2.4.2 加强for循环（Enhanced for Loop）

JDK 1.5 引进了一种新的循环类型，被称为For-Each循环或者加强型循环，它能在不使用下标的情况下遍历数组。
语法格式如下。

```
for(type element: array){
}
```

例如下面的代码可以输出数组里的所有元素。

```
for(double value: myList){
    System.out.println(value);
}
```

它与下面的for循环代码效果是一样的。

```
 for (int i = 0; i < myList.length; i++) {
    System.out.println(myList[i]);
 }
```

加强for循环主要用于顺序读取数组或集合中的元素，而不适用于需要索引来访问元素的场景，因此也无法修改数组或集合中的元素。就比如前面的将数组里的元素移动的操作无法进行。再比如下面的代码。

```
int[] myList = new int[]{1, 2, 3, 4, 5};
for (int number : myList) {
    number = number * 2;
}
for (int number : myList) {
    System.out.println(number);
}
//输出的结果还是1 2 3 4 5
```

我们可以理解成for-each循环中的number时原来数组的副本，因此对number的修改不会影响原数组myList中的元素。

2.4.3 多维数组

多维数组可以看成是数组的数组，比如二维数组就是一个特殊的一维数组，其每一个元素都是一个一维数组，格式如下。

```
type[][] typeName = new type[typeLength1][typeLength2];
```

type 可以为基本数据类型和复合数据类型（类和对象，将在后面介绍），typeLength1 和 typeLength2 必须为正整数，typeLength1 为行数，typeLength2 为列数。
例如下面的代码。

```
int[][] a = new int[2][3];
```

这个代码中创建了一个二维数组a，可以看成是一个两行三列的数组。
对二维数组中的每个元素，引用方式如下。

```
arrayName[index1][index2]
```

我们可以对二维数组的每一维分开分配不同长度的空间，如下所示。

```
String[][] s = new String[2][];
s[0] = new String[2];
s[1] = new String[3];
s[0][0] = new String("Good");
s[0][1] = new String("Luck");
s[1][0] = new String("to");
s[1][1] = new String("you");
s[1][2] = new String("!");
```

这串代码中，我们先创建了一个包含两个元素的二维数组s，其中每个元素是本身也是一个String数组。s[0]是第一行，它是一个长度为2的String数组。s[1]是第一行，它是一个长度为3的String数组。后面再分别给每个元素赋值。其中我们就用前面说的引用方式对元素进行了访问。

2.5 总结

需要注意的点主要有4点：
1.数组这种数据结构只能存储在声明了数据类型后只能存储对应数据类型的数据。
2.数组一旦被创建，其大小是固定的，不能动态增加或减少，后面将会以这点介绍其他的数据结构。
3.数组的索引范围是从0开始，而且索引值不能超过上限，即最大只能是数组的长度-1，这个是作为程序

员经常需要注意的。
4.数组是引用类型（Reference Types）变量，而不是基础数据类型（Primitive Types）变量。即变量存储的是对象的引用（内存地址），而不是对象本身。
两者的区别可以看下面两段代码。

```
int a = 10;
int b = a;
a = 20;
System.out.println(b); // 这里print的结果是10，而不是20
```

```
int[] array1 = {1, 2, 3};
int[] array2 = array1;
array1[0] = 10;
System.out.println(array2[0]);// 这里print结果是10，因为array2指向了array1的地址
```

因此如果array2原来有值，在array2 = array1;后array2原来的值就会被丢弃，要想复制一个数组可以使用下面的几种方法。
方法一：使用循环。

```
 int[] sourceArray={2, 3, 1, 5, 10};
 int[] targetArray=new int[sourceArray.length];
 for (int i = 0; i < sourceArray.length; i++){
    targetArray[i] = sourceArray[i];
 }
```

方法二：使用arraycopy方法

```
System.arraycopy(sourceArray, src_pos, targetArray, tar_pos, length);
```

其中sourceArray是被复制的数组。
src_pos是被复制的数组的起始位置，表示从该位置开始复制数据。
targetArray是目标数组，表示要被复制到这个数组中。
tar_pos是目标数组中的起始位置，表示从该位置开始粘贴数据。
length表示要复制的元素数量。

因此实践起来如下。

```
int[] sourceArray={2, 3, 1, 5, 10};
int[] targetArray=new int[sourceArray.length];
System.arraycopy(sourceArray, 0, targetArray, 0, sourceArray.length);
```

2.6 进阶实践

我们通过一些进阶实践来进一步查看Arrays的使用顺便复习一下其他Java的知识点。

2.6.1 在方法中使用数组作为参数

下面的代码使用数组作为参数。

```
 public static void printArray(int[] array) {
    for (int i = 0; i < array.length; i++) {
         System.out.print(array[i] + " ");
    }
 }
```

我们想使用这个方法时可以提前创建一个数组，然后传入这个数组。如下列代码所示。

```
int[] list = {3, 1, 2, 6, 4, 2};
printArray(list);
```

如果这个数组我们只需要使用一次，我们也可以使用匿名数组（Anonymous Array）即这个数组没有变量名（显式的引用变量），因此我们无法在别的地方再使用它。匿名数组是在调用方法时直接创建的数组。
示例如下。

```
printArray(new int[]{3, 1, 2, 6, 4, 2});
```

说到方法，Java 使用值传递的方式来将参数传递给方法。
对于引用类型数据来说，参数的值是将引用类型数据的引用（内存地址）传递给方法，在方法内部对数组所做的任何更改都会影响到作为参数传递的原始数组。
对于基础数据类型来说，参数是按值传递的，即实际的值被传递。在方法内部更改局部参数的值不会影响方法外部变量的值。实际上传递的是该值的一个副本，方法内部的参数被视为局部变量。
下面代码演示了两者额区别，这里使用了方法重载（Overloading）即在同一个类中，可以有多个同名方法，只要它们的参数列表不同即可。参数列表的不同可以是参数的个数不同，参数类型不同，或者参数的顺序不同。

```
public class Main {
    public static void main(String[] args) {
        int num = 1;
        System.out.println(num);// 输出1
        add(num);
        System.out.println(num);// 输出1，没有修改
        int[] numbers = new int[]{1};
        System.out.println(numbers[0]);// 输出1
        add(numbers);
        System.out.println(numbers[0]);// 输出3，修改
    }
    public static void add(int num){
        num = num + 2;
    }
    public static void add(int[] num){
        for (int i = 0; i < num.length; i++) {
            num[i] = num[i] + 2;
        }
    }
}
```

这里的进一步解释是，由于Java虚拟机（JVM）使用堆（Heap）内存来存储数组和对象，堆内存用于动态内存分配，其中内存块可以以任意顺序分配和释放。当方法调用这个数组是，这里传递的就是该数据的引用（内存地址），因此对它的修改就会作用与原数组上。

2.6.2 方法返回数组

前面我们使用void关键字，即该方法执行后不返回任何结果，现在我们尝试返回一个数组。
下面的示例展示了如何将一个数组翻转。

```
public static int[] reverse(int[] list) {
    int[] result = new int[list.length];
    for (int i = 0, j = result.length - 1; i < list.length; i++, j--) {
        result[j] = list[i];
    }
    return result;
}
```

这里for循环中使用了i和j两个参数，当然你也可以使用一个参数去解决。

```
public static int[] reverse(int[] list) {
    int[] result = new int[list.length];
    for (int i = 0; i < list.length; i++) {
        result[result.length - 1 - i] = list[i];
    }
    return result;
}
```

当然这样做返回了一个新的数组，这么做不会对原来的数组进行修改。如果需要我们对原数组进行翻转，我们可以利用前面说的知识，示例如下。

```
public static void reverse(int[] list) {
    int temp;
    for (int i = 0, j = list.length- 1; i < list.length/2; i++, j--) {
        temp = list[j];
        list[j] = list[i];
        list[i] = temp;
    }
}
```

同理一个参数的示例如下。

```
public static void reverse(int[] list) {
    int temp;
    for (int i = 0; i < list.length/2; i++) {
        temp = list[list.length - 1 - i];
        list[list.length - 1 - i] = list[i];
        list[i] = temp;
    }
}
```

2.6.3 在数组中查询某元素

现在我们需要查找数组里的某个元素，返回第一次出现的这个元素的index值，如果没有这个元素，我们便返回-1。
一个很简单的思路便是我们一个个去对照，如果找到便返回这里的index值，如果没有找到便返回-1。我们叫这种方式为线性搜索（Linear Search）。

```java
public static int linearSearch(int[] list, int key) {
    for (int i = 0; i < list.length; i++){
        if (key == list[i]){
            return i;
        }
    }
    return -1;
}
```

之前我们在INT102这门课中学习了一些基础的算法，这些算法都是对刚刚的线性搜索的升级。下面我们在Java中实践一下其中的几个算法。
先是二分搜索（Binary Search），二分搜索的前提是这个数组里的元素是提前排序好的，下面假设这个数组是从低到高排序的。先将目标元素与中间的元素进行对比，如果相等，那就返回该元素的index，如果目标元素小于中间元素，则去前半部分中继续寻找，否则就去后半部分中继续寻找。然后再不断重复这个过程，如果搜寻完还是没有找到，那就返回-1。
代码如下。

```java
public static int binarySearch(int[] list, int key) {
    int low = 0;
    int high = list.length- 1;
    int mid;
    while (high >= low) {
        mid = (low + high) / 2;
        if (key < list[mid]){
            high = mid - 1;
        }
        else if (key == list[mid]){
            return mid;
        }
        else{
            low = mid + 1;
        }
    }
    return -1;
}
```

它的时间复杂度是$O(logn)$。
Java其实自带了几个重载的binarySearch方法，这些方法可以在int、double、char、short、long和float类型的数组中搜索键。

```java
int[] list = {1, 2, 3, 4, 6, 7, 8, 9};
System.out.println("Index is " + java.util.Arrays.binarySearch(list, 6));// 输出是4
```

2.6.4 给数组排序

前面说了二元搜索需要数组内的元素是排序好的，那么如何给数组排序呢。
下面几种排序方式，这也是INT102中学习过的。

2.6.4.1 选择排序（Selection Sort）

这个方法是将剩余元素中的最小（或最大）元素，存放到序列的起始位置，直到全部元素被排序。

代码主要是两部分，首先找到最小（或最大）元素，然后将其放到对应的位置，细节如下。

```
public static void selectionSort(double[] list) {
    for (int i = 0; i < list.length; i++) {
        // Find the minimum in the list[i..list.length-1]
        double currentMin = list[i];
        int currentMinIndex = i;
        for (int j = i + 1; j < list.length; j++) {
            if (currentMin > list[j]) {
                currentMin = list[j];
                currentMinIndex = j;
            }
        }
        // Swap list[i] with list[currentMinIndex] if necessary;
        if (currentMinIndex != i) {
            list[currentMinIndex] = list[i];
            list[i] = currentMin;
        }
    }
}
```

2.6.4.2 插入排序（Insertion Sort）

这个方法是不断构件有序序列，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入。

代码需要注意的是我们比较的时候，如果未排序的元素直接大于排序好的元素的最大值就无需插入，而对于需要插入的元素，插入的位置知道后，要将插入位置后面的元素向右移动一个位置，细节如下。

```
public static void insertionSort(int[] a){
    for(int i=1; i<a.length; i++){
        int temp = a[i];
        if(temp < a[i-1]) {
            int j;
            for(j = i-1; j >= 0; j--){
                if(temp < a[j]){
                    a[j+1] = a[j];
                }
                else{
                    break;
                }
                a[j+1] = temp;
            }
        }
    }
```

```
}
```

同理Java也有内置的插入排序方法在java.util.Arrays类中。示例如下。

```
double[] numbers = {6.0, 4.4, 1.9, 2.9, 3.4, 3.5};
java.util.Arrays.sort(numbers);
char[] chars = {'a', 'A', '4', 'F', 'D', 'P'};
java.util.Arrays.sort(chars);
```

3. Classes（类）

类（Class）是 Java 中实现面向对象编程（Object-Oriented Programming，OOP）的核心概念之一，它是Java中面向对象编程最好的体现，它为创建对象提供了模板或蓝图。
在面向对象编程中，我们使用对象表示现实世界中的一个实体，它可以从具有共同属性（properties）的类/对象模板中被明确识别出来。在Java中，我们使用类来定义相同类型对象。类描述了一组属性（数据字段）和行为（方法），用于创建对象。

一个对象有自己的状态（state）和行为（behavior）。
1.状态是由一些数据字段（或者说属性）机器当前值组成。
2.行为是由一组实例方法定义。

3.1 类的组成

因此Java的类由三部分组成。
1.非静态/实例变量（Instance Variables）：用于定义对象的数据字段。这些变量是每个对象独有的，存储对象的状态。
2.非静态/实例方法（Instance Methods）：用于定义对象的行为。实例方法是对象可以执行的操作，这些方法只能通过类的实例（对象）来调用，而不能通过类本身直接调用。
3.构造方法（Constructors）：一种特殊类型的方法，用于创建类的实例（对象）。构造方法在创建对象时被调用，用于初始化对象的状态。

例如我现在有一个类的名字为Human，其中有一个实例方法addAge以增加年龄，其中我创建了一个名为Peter的Human对象。

```
public class Human {
    // 实例变量（数据字段）
    int age;

    // 构造方法
    public Human() {
        this.age = 0;
    }
    public Human(int age) {
        this.age = age;
    }

    // 实例方法
    public void addAge(int years) {
        age += years;
        System.out.println("Age updated to: " + age);
    }
}
```

```
public class Main {
    public static void main(String[] args) {
         Human Peter = new Human(30); // 创建Human类的实例Peter
         Peter.addAge(5); // 正确调用：通过实例（对象）调用实例方法
         // Human.addAge(5); // 错误调用：尝试通过类直接调用实例方法
         Human Alice = new Human();
         Alice.addAge(5);
    }
}
```

我可以使用Peter调用addAge方法，但是不能使用Human调用addAge方法。
我们在构造方法中经常见到this关键字，它是用于引用当前对象的实例变量，即上面的构造方法中我们接受了一个参数age，但是该类本来就有一个名为age的实例变量，因此用this.age表示当前对象的实例变量以区分参数的age。
此外上面的代码中构造方法有两种方法，其中上面的那个是无参构造方法，下面的那个需要接受一个参数从而将这个参数用以赋值。其实如果我们如果不定义任何构造方法，那编译器会自动提供一个无参构造方法作为默认构造方法，这个方法没有参数，并且不执行任何操作。这个默认构造方法会将类的实例变量初始化为默认值。规则与前面说的基础数据类型和引用类型的默认值一致。
说到构造方法，首先构造方法必须与类名相同。构造方法没有返回类型，虽然它的作用是初始化新创建的对象。构造方法和前面学习的创造一个数组类似，使用new操作符以初始化对象，以便对象可以引用变量。我们可以理解成先声明了对应的数据类型（即类名），然后变量名，用new生成对应的对象，然后将对象的引用（内存地址）赋值给刚刚的变量名。相关的语法格式如下。

```
ClassName o = new ClassName();
```

例如前文的。

```
Human Peter = new Human(30);
Human Alice = new Human();
```

当我们想访问对象的属性时，方法和前面调用实例方法类似，我们使用对象的引用变量访问对象的状态（属性）和行为（方法）。
格式如下。

```
// 访问状态
objectRefVar.data
// 访问行为
objectRefVar.methodName(arguments)
```

示例如下。

```
Alice.age;
Alice.addAge(5);
```

3.2 面向对象设计

我们上学期学习了CPT203软件工程，其中学习的UML（Unified Modeling Language，统一建模语言）可以提供一种可视化面向对象系统设计的标准方法。

一般我们使用类图用来设计面向对象编程。

其中对于不同访问修饰符，类图的表示方式有所差异。
public表示类、数据或方法对任何包中的任何类都是可见的。在UML中用\\+//表示。
private表示数据或方法只能被声明它们的类访问。在UML中用\\－//表示。
对于private修饰的数据我们使用访问器（accessors）和修改器（mutators)方法来读取和修改私有属性。语法格式如下。

```
// 访问器
getField();
// 修改器
setField();
```

下图展示了一个UML的例子。

3.2.1 包和修饰符

这里对访问权限作进一步解释。
private 修饰符：限制访问权限仅限于类内部。
默认修饰符：限制访问权限仅限于包内部。
public 修饰符：无限制访问权限。
下图展示了两个例子。

上面一个例子中，在pakage p1中，C1类有一个公共属性x，一个默认属性y，一个私有属性z。C2类可以访问C1的公共属性x，以及默认属性y，但不能访问私有属性z。C2类可以调用C1的m1()和m2()但不能调用m3()。对于package p2中，C3类不能访问C1类的y和z，也不能调用m2()和m3()。
在下面的例子中，C1和C2都在p1包内，所以C2可以直接访问C1类，但是对于在p2包内的C3来说，C3只能访问C2，不能访问C1。

3.3 静态（Static）与非静态（non-static）变量

静态变量（Static variables）是类级别的变量，属于类本身，而不是类的某个特定实例（对象），因此它在整个类的所有实例之间共享，对静态变量的修改也会影响到所有实例。
例如对于前面的例子中，我们可以添加一个静态变量用于记录这里目前有多少人。

```
public class Human {
    // 静态变量
    static int count = 0;
    // 实例变量（数据字段）
    int age;

    // 构造方法
    public Human() {
        this.age = 0;
        count++; // 每创建一个实例，计数增加1
    }
    public Human(int age) {
        this.age = age;
        count++; // 每创建一个实例，计数增加1
    }

    // 实例方法
    public void addAge(int years) {
        age += years;
        System.out.println("Age updated to: " + age);
```

```
    }

    // 静态方法用于获取当前人类数量
    public static int getCount() {
        return count;
    }
}

public class Main {
    public static void main(String[] args) {
         Human Peter = new Human(30); // 创建Human类的实例Peter
         Peter.addAge(5); // 正确调用：通过实例（对象）调用实例方法
         // Human.addAge(5); // 错误调用：尝试通过类直接调用实例方法
         System.out.println("Number of humans: " + Human.getCount()); // 输出当前人类数量，结果为1

         Human Alice = new Human();
         Alice.addAge(5);
         System.out.println("Number of humans: " + Human.getCount()); // 输出当前人类数量，结果为2
    }
}
```

我们也可以在一个圆的类中，用静态变量以定义圆周率以供圆里的每个对象使用，而且像这种我们不会修改的常量，可以再使用final关键字以防止被修改。如下所示。

```
static final double PI = 3.1415926;
```

上面的示例中，我们还创建了一个名为getCount的静态方法。静态变量（也称为类变量）和静态方法属于类本身，而不是类的某个特定实例（对象）。因此访问静态变量或调用静态方法时，使用类名，而不是实例名。

```
System.out.println("Number of humans: " + Human.getCount());
```

因此这里是Human.getCount()。

3.4 默认值与值传递问题

前面我们说过Java会给数据字段分配默认值。
对于基础数据类型（Primitive Data Types）：
byte：默认值为 0
short：默认值为 0
int：默认值为 0
long：默认值为 0L
float：默认值为 0.0f
double：默认值为 0.0d
char：默认值为 \u0000（即空字符）
boolean：默认值为 false
其余的默认值为null
注意如果一个类的实例变量是另一个类的实例，那么这个变量会继承或使用那个类中定义的默认值。

需要注意的是Java 不会给方法内部的局部变量分配默认值。
比如下面的代码会提示你尚未给变量进行初始化。

```
public class Test {
    public static void main(String[] args) {
        int x; // x has no default value
        String y; // y has no default value
        System.out.println("x is " + x);
        System.out.println("y is " + y);
    }
}
```

由于类也属于一种引用类型。因此我们要是想复制一个对象和前面的数组一样（它们都是引用类型，或者详细点说数组是java.lang.Object类的子类，即java.lang.reflect.Array类的实例），不能直接使用c1 = c2这种方式。
当一个对象不再被任何引用变量引用时，它就变成了垃圾（garbage）。JVM会自动将这些垃圾回收，即自动释放不再被引用的对象所占用的内存，以便这些内存可以被重新使用。
在一些旧的编程语言中，如C和C++，程序员需要手动管理内存，即显式地分配和删除不再使用的数据或对象。在Java中，垃圾回收机制自动处理不再需要的对象，减轻了程序员的负担。

3.5 实用类

Java有许多实用类，下面介绍其中的几个。

3.5.1 Date

java.util.Date类提供系统独立的日期和时间封装（Encapsulation，封装指的是将对象的状态（属性）和行为（方法）捆绑在一起，并对外隐藏对象的内部实现细节，只暴露有限的访问接口，我们前面做的有关类的操作都是封装），其中的toString()用于返回表示日期和时间的字符串。

```
java.util.Date date = new java.util.Date();
System.out.println(date.toString());
```

这个代码创建了一个表示当前日期和时间的Date对象，然后将日期和时间转换为字符串形式呈现了出来。
但java.util.Date类已经被Java 8中的java.time包中的新日期时间API所取代，如LocalDateTime、ZonedDateTime等。
其余的使用可以看下面这张图。

3.5.2 Random

我们平时使用的Random也是通过java.util.Random类实现的。
细节如下图所示。

相关的示例代码如下所示。

```
Random random1 = new Random(3);
for (int i = 0; i < 10; i++)
    System.out.print(random1.nextInt(1000) + " ");
```

我们先创建了Random类的实例random1，然后将其初始化为使用种子值3。然后调用Random实例方法nextInt生成一个介于0（包含）和1000（不包含）之间的随机整数。

3.6 对象数组

对象数组是引用变量的数组，类似于之前见过的多维数组。我们前面说过声明数组时的type可以是复合数据类型，即这里的类和对象。
对象数组意味着数组中的每个元素都是指向某个对象的引用。
例如我们可以创建一个Human类型的对象数组humanArray，如下所示。

```
Human[] humanArray = new Human[2];
humanArray[0] = new Human();
humanArray[1] = new Human(5);
```

4. 练习

我们通过一些练习来复习本章学习的知识。

4.1 基础练习

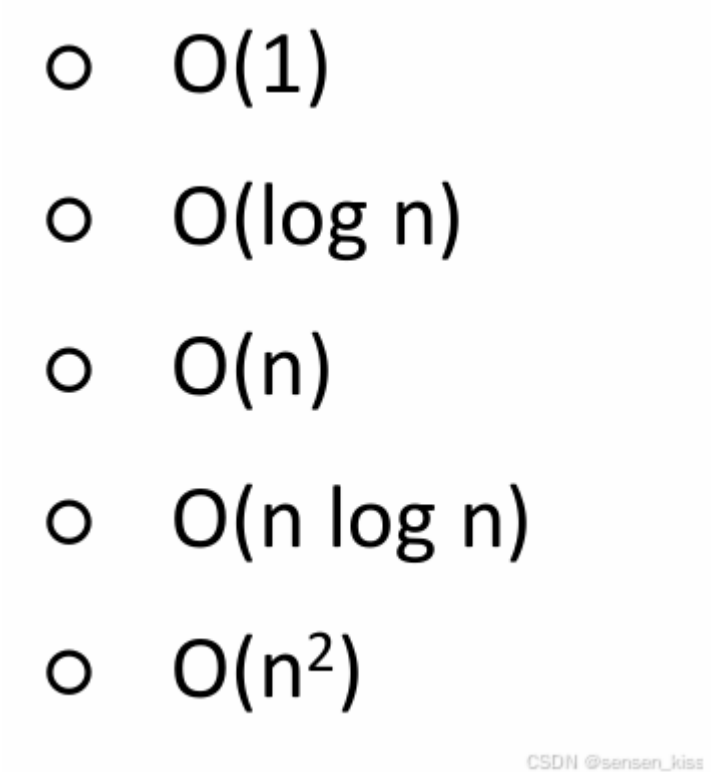
1.下图中哪一个是有效的初始化数组方法?

A.错误的，正确格式为int[] arr = new int[]{1, 2, 3, 4, 5};
B.错误的，正确格式参考C。
C.正确的。
D.错误的，正确格式参考C。

2.下图的结果应该是哪一个?

答案是B，因为for-each循环无法修改值。

3.二分搜索的最坏情况下的时间复杂度是多少?



CSDN @sensen_kiss

答案是B。

4.对于以下代码我该如何输出x和y的值呢?

x可以使用new InClassQuiz().x而y可以使用InClassQuiz.y，即一个是实例变量，需要通过实例进行访问，而另一个是静态变量，需要通过类进行访问。

5.下列代码的输出结果是多少?

答案是10，代码中将obj1的引用传给了obj2，所以obj2现在也指向obj1的值，因此修改obj2.x的值会影响obj1的值，因此最后的结果是10。

4.2 进阶练习

按照下面的要求完成相关代码。

4.2.1 将用户输入的数组进行翻转

1.程序问用户想要输入多少个数字。
2.程序将用户输入的数字全部以Double数据类型的数组存储起来。
3.程序将这个数组用一个名为reverseInPlace方法翻转后呈现给用户。

参考代码如下。

```
import java.util.Scanner;

public class Exercise_1_1 {
    public static void reverseInPlace(double[] arr) {
        for (int i = 0; i < arr.length / 2; i++) {
            double temp = arr[i];
            arr[i] = arr[arr.length - 1 - i];
            arr[arr.length - 1 - i] = temp;
        }
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the size of array: ");
        int size = Integer.parseInt(sc.nextLine());
        double[] arr = new double[size];

        System.out.println("Enter the elements of array: ");
        for (int i = 0; i < size; i++) {
            arr[i] = sc.nextDouble();
        }

        reverseInPlace(arr);

        System.out.print("The reversed array is: [");
        for (int i = 0; i < size; i++) {
            System.out.print(arr[i] + " ");
        }
        System.out.println("]");
    }
}
```

4.2.2 判断数组是否含有连续四个相同的值

1.程序使用public static boolean isConsecutiveFour(int[] values)方法以判断数组是否含有连续四个相同的值。
参考代码如下。

```
import java.util.Scanner;

public class Exercise_1_2 {
    public static boolean isConsecutiveFour(int[] values){
        int len = values.length;
        if(len < 4){
            return false;
        }
        else{
            for (int i = 0; i < len - 3; i++){
                if (values[i] == values[i + 1] && values[i] == values[i + 2] &&
values[i] == values[i + 3]){
                    return true;
                }
            }
            return false;
        }
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the number of elements in the array: ");
        int size = Integer.parseInt(sc.nextLine());
        int[] arr = new int[size];
        System.out.println("Enter the elements in the array: ");
        for (int i = 0; i < size; i++) {
            arr[i] = sc.nextInt();
        }
        boolean isConsecutive = isConsecutiveFour(arr);
        if(isConsecutive){
            System.out.println("Consecutive");
        }
        else {
            System.out.println("Not Consecutive");
        }
    }
}
```

4.2.3 用于计算运行时间的类

设计一个名为StopWatch的类。
该类包含：
1.带有getter方法的私有数据字段startTime和endTime。
2.一个空构造函数，用当前时间初始化startTime和endTime。
3.一个名为start()的方法，它将startTime重置为当前时间。
4.一个名为stop()的方法，将endTime设置为当前时间。
5.一个名为getElapsedTime()的方法，它返回经过的时间（秒表的停止和开始时间（以毫秒为单位）。

可以使用System.currentTimeMillis()来获取当前时间。
最后编写一个测试程序，测量使用讲座中讨论的选择排序对100000个数字进行排序的执行时间。
参考代码如下。

```
public class Exercise_1_3 {
    public static class StopWatch{
        private long startTime;
        private long endTime;

        public StopWatch(){
            startTime = System.currentTimeMillis();
            endTime = startTime;
        }

        public void start(){
            startTime = System.currentTimeMillis();
        }
        public void stop(){
            endTime = System.currentTimeMillis();
        }
        public double getElapsedTime(){
            return (endTime - startTime) / 1000.0;
        }
    }


    public static void selectionSort(int[] list) {
        for (int i = 0; i < list.length; i++) {
            // Find the minimum in the list[i..list.length-1]
            int currentMin = list[i];
            int currentMinIndex = i;
            for (int j = i + 1; j < list.length; j++) {
                if (currentMin > list[j]) {
                    currentMin = list[j];
                    currentMinIndex = j;
                }
            }
            // Swap list[i] with list[currentMinIndex] if necessary;
            if (currentMinIndex != i) {
                list[currentMinIndex] = list[i];
                list[i] = currentMin;
            }
        }
    }

    public static void main(String[] args) {
        int[] numbers = new int[100000];
        for(int i = 0; i < numbers.length; i++){
            numbers[i] = (int)(Math.random() * 100000);
        }
        StopWatch stopWatch = new StopWatch();
        stopWatch.start();

        selectionSort(numbers);
        stopWatch.stop();
```

```
        System.out.println(stopWatch.getElapsedTime());
    }
}
```

4.2.4 学生记录类

创建一个学生记录类，代表学生参加高级OOP课程。
1.每个学生对象应代表名字、姓氏、电子邮件地址和组号。
2.包含一个toString()方法，该方法返回一个学生的字符串表示形式，以及一个less()方法通过两个学生的组号进行比较。
最后编写一个测试程序，打印和比较学生对象。
参考代码如下。

```
public class Exercise_1_4 {
    public static class Student {
        private float firstName;
        private float lastName;
        private float emailAddress;
        private int groupID;

        public Student(float firstName, float lastName, float emailAddress, int
groupID) {
            this.firstName = firstName;
            this.lastName = lastName;
            this.emailAddress = emailAddress;
            this.groupID = groupID;
        }

        public String toString(){
            return "Student: " + firstName + " " + lastName + " " + emailAddress
+ " " + groupID;
        }
        public void less(Student other){
            if (this.groupID > other.groupID){
                System.out.println("This student is bigger");
            }
            else if (this.groupID < other.groupID){
                System.out.println("This student is smaller");
            }
            else {
                System.out.println("They are equal");
            }
        }

    }
    public static void main(String[] args) {
        Student student1 = new Student(1,2,3,4);
        Student student2 = new Student(2,3,4,5);
        Student student3 = new Student(3,4,5,4);
        student1.less(student2);
        student2.less(student3);
        student1.less(student3);
    }
}
```