Session 4

Table of Contents

Revision

1. A class is a blueprint for defining a real or abstract concept. Examples of classes are Person, Student, Employee, Shape, Stack, etc.
2. An instance variable is a property of each instance of the class. For example, *each* Person instance will have a name (String), an age (int), and height (double). these properties typically distinguish one instance of a class from another instance.
3. A static variable is a property of the class as a whole. All instances of the class share the static variable. eg: Math.PI
4. An instance method is a function that belongs to an instance of the class. For example, *each* instance of the Person class will have a method called growOld() or drinkComplan();
5. A static method is a function that belongs to the class as a whole. This is typically done when it doesn't make sense for each instance to have the method. For example, functions like GCD, isPrime, etc do not belong to any one instance of any class.
6. The final keyword prevents the variable from being assigned a value after it has been initialized.

Example:

```
class Person {
        // instance variables
        String name;
        int age;
        double height;

        final int yearOfBirth;// Because obviously your birth year cannot change

        // Parameterized constructor
        Person(String n, int a, double h, int y) {
                name = n;
                age = a;
                height = h;
                yearOfBirth = y;
        }

        // Instance method
        void drinkComplan() {
                height += 0.1;
        }
}

class ComplanAdvertisement {
        public static void main(String[] args) {

                Person p = new Person("Peter", 18, 5.1, 1999);
                System.out.println(p.getHeight());              //      Prints out 5.1

                p.drinkComplan();
                System.out.println(p.getHeight());              //      Prints out 5.2
        }
}
```

Initializer Blocks

There are two types of initializer blocks in Java.

1. Instance Initializer Block
   As the name suggests, it is used to initialize instance variables. Example:

```
class Person {
        int age;

        {
                age = 18;
        }
}
```

The initializer block's content is copied into **every** constructor by the compiler. This means that you can initialize final variables in the instance initializer block as well.

2. Static Initializer Block
   This is executed once for each class.

```
class MathExample {
        static double pi;

        static {
                pi = 3.14;
        }
}
```

You are allowed to put arbitrary statements inside initializer blocks. Initializer blocks are a *slightly* advanced concept, and so it is sufficient for you to understand and remember the above, without worrying too much about internal details of their working.

Call Stack

The call stack is a memory structure maintained by the JVM in order to keep track of function calls. Almost EVERY language has the concept of a call stack, since almost every language has the concept of functions.

Whenever a function is called, a stack "frame" is created, and pushed on the stack. This is simply a fancy term for a structure that can hold all the local variables and arguments passed to the function. When the function returns, the frame is popped off.

Some important points:

1. The call stack is used for ALL function calls, not just recursive calls.
2. If you call too many functions (typically happens in infinite/deep recursion), the stack memory fills up, and you get a StackOverflowError
3. **Each thread has its own call stack.** Whenever you start a new thread, a new call stack is created from that point on.

Object Heap

The Object Heap is a memory structure maintained by the JVM to store objects, i.e. instances of classes. Everything other than the 8 primitive data types are objects, including Strings and arrays.

It is important to understand the difference between objects and object references:

1. *Objects* are what are created by the `new` keyword, and are stored on the heap **always**.
2. *Object references* are "pointers" to the objects on the heap. They may be on the call stack (if created inside a function), or on the heap (if they are a instance variable of another class.)

Some Traps and Pitfalls

1. String

   The String class is treated slightly specially in the Java language. In particular, Strings are cached (or "memoized") by the compiler. These cached values are stored in a location on the heap called the *string pool*. For example, in the following snippet:

   ```
   String s1 = "hello";
   String s2 = "hello";
   ```

   you'd expect two different string objects to be created on the heap. However, only one is created, i.e. s1 and s2 point to the same object on the heap.

2. Wrapper classes

   A small cache is maintained for small valued integers as well, similar to the String pool. So if you make 2 `Integer` objects with the value 5, it is highly likely that i1 == i2 will be true, i.e. they will end up pointing to the same object in memory.

   You need not know too much about this concept, given that it is slightly advanced, and typically does not play an important role in day to day development.

Recursion And Memoization Practice

Write recursive and memoized versions of the following sequences:

1. Tribonacci Series
   $T_0 = 0$
   $T_1 = 1$
   $T_2 = 1$
   $T_n = T_{n-1} + T_{n-2} + T_{n-3}$

2. Lucas Numbers
   $L_0 = 2$

   $L_1 = 1$
   $L_n = L_{n-1} + L_{n-2}$

3. Pell Numbers
   $P_0 = 0$
   $P_1 = 1$
   $P_n = 2*P_{n-1} + P_{n-2}$

4. Pascal's Triangle
   ${}^nC_0 = 1$
   ${}^nC_n = 1$
   ${}^nC_r = {}^{n-1}C_r + {}^{n-1}C_{r-1}$

Assume the following input format for the first 3 sequences:

1. The first line is an integer T, the number of test cases
2. T integers follow. For each integer n, print on a single line, space separated, the n^{th} Tribonacci, Lucas and Pell number.
   Verify your work by checking the sequence values on the links.

BufferedReader - The Faster Way Of Accepting Input

As you may have noticed when solving last session's String problems, Scanner has a slight drawback, i.e. it does not play well when you intertwine calls of sc.next() and numeric input (sc.nextInt(), sc.nextDouble(), etc)

In addition, it also has the drawback of being slightly slow. When you start solving problems on CodeChef, using Scanner will almost always result in timeouts. As a replacement, you can use BufferedReader. Below is a complete example of how to use BufferedReader to accept and transform numerical input:

Imagine that the input format is:

1. The first number is the number of test cases T, 1 <= T <= 100
2. 2*T lines follow, 2 lines per test case.
   i. The first line is the number of elements in the array
   ii. The second is an array of integers.

```
class BufferedReaderExample {
    // Note the "throws IOException"
    // Not writing those lines will end up in a compiler error
    // We will see Exceptions in detail in a future session
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        // br.readLine() returns a single String, which is the current line of input
        String s = br.readLine();
        int T = Integer.parseInt(s);

        // Similar methods are Long.parseLong(), Double.parseDouble(), etc

        while (T != 0) {
            // You can do it in a single line as well
            int n = Integer.parseInt(br.readLine());

            int[] arr = new int[n];

            // br.readLine() would result in a String which looks something like "1 2 3 10 5 4"
            // Split returns an array of strings, depending on the delimiter, which is mostly a space
            // sa now has {"1", "2", "3", "10", "5", "4"}
            String[] sa = br.readLine().split(" ");

            for (int i = 0; i < n; i++) {
                arr[i] = Integer.parseInt(sa[i]);
            }
```

```
                    // Do usual stuff with arr

                    T--;
                }
            }
        }
```

Sorting - A High Level Perspective

Sorting is simply ordering a collection according to natural ordering, or some comparator logic. For instance, an array of integers can be sorted in ascending order, an array of Strings can be sorted in lexicographical (dictionary) order, and so on.

There are dozens of sorting algorithms, which we will see in a later session. For now, remember that very rarely will you have to write a sorting algorithm from scratch. It is only in certain problems where you have to modify an existing sorting algorithm to get to the solution. In most other cases, if you want to sort an array (or any other collection), it is best to use in built methods since they have been scrutinized and optimized far beyond what a hand-written algorithm would be capable of.

In Java, you can use `Arrays.sort` static (overloaded) method from the Arrays utility class from the `java.util` package.

Assignments

HackerRank

1. https://www.hackerrank.com/challenges/hackerrank-in-a-string
2. https://www.hackerrank.com/challenges/weighted-uniform-string
3. https://www.hackerrank.com/challenges/red-john-is-back
4. https://www.hackerrank.com/challenges/stockmax (HARD)
5. https://www.hackerrank.com/challenges/minimum-absolute-difference-in-an-array
6. https://www.hackerrank.com/challenges/grid-challenge
7. https://www.hackerrank.com/challenges/luck-balance
8. https://www.hackerrank.com/challenges/maximum-perimeter-triangle
9. https://www.hackerrank.com/challenges/priyanka-and-toys
10. https://www.hackerrank.com/challenges/mark-and-toys
11. https://www.hackerrank.com/challenges/angry-children

Miscellaneous

TODO

Test

Link: https://www.hackerrank.com/contests/c2c2017-3