built on 2023/04/26 at 15:16:06 ***due:*** *wed may 3 @ 11:59pm*

This assignment will introduce you to basic Java constructs. You will implement a few Java programs, test them thoroughly, and hand them in. It's imperative that you use Java 11.

For each task, write helper functions as you see fit. Strive to write code that promotes clarity.

Overview:

Problem	File Name
1.	`workspace.jpg`
2.	`Diamond.java`
3.	`Roman.java`
4.	`Fib.java`

Problem	File Name
5.	`Subsel.java`
6.	`Happy.java`
7.	`Hidden.java`

Collaboration

We interpret collaboration very liberally. You may work with other students. However, each student ***must*** write up and hand in his or her assignment separately. Let us repeat: You need to write your own code. You must not look at or copy someone else's code. You need to write up answers to written problems individually. The fact that you can recreate the solution from memory will be taken as proof that you actually understood it, and you may actually be interviewed about your answers.

Be sure to indicate who you have worked with (refer to the hand-in instructions).

Logistics

We're using a script to grade your submission before any human being looks at it. Sadly, the script is not as forgiving as we are. *So, make sure you follow the instructions strictly.* It's a bad omen when the course staff has to manually recover your file because the script doesn't like it. Hence:

- Save your work in a file as described in the task description. This will be different for each task. **Do not save your file(s) with names other than specified.**
- You'll zip these files into a single file called `a1.zip` and you will upload this one zip file to Canvas before the due date.
- Attempt to solve each task on your own first. After a day, if you still can't solve a task, come to office hours.
- Before handing anything in, you should thoroughly test everything you write.
- The course staff is here to help. We'll steer you toward solutions. Catch us in real-life or online.

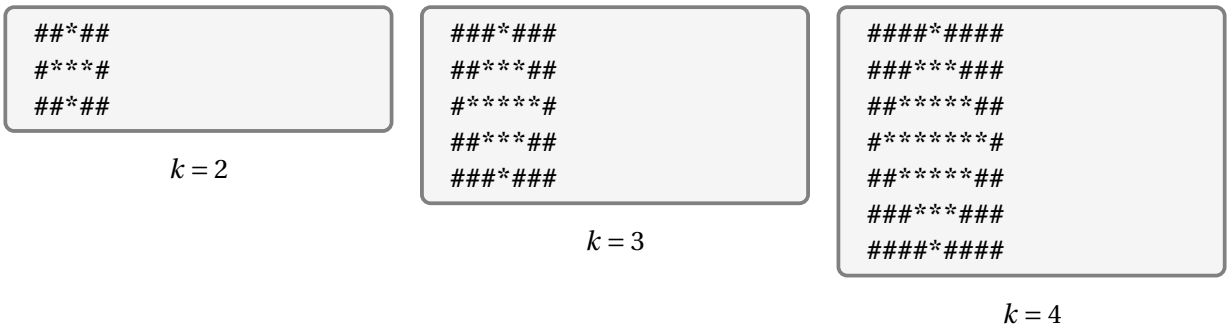

Figure 1: Examples of diamonds of different sizes.

Task 1: Install Java 11 and an IDE on your computer (2 points)

In the first few weeks of this class, we'll be using a text/code editor to write Java programs. We recommend IntelliJ and VSCode (both work across various platforms).

What to Hand In? You are handing in one JPEG file called `workspace.jpg`, featuring on the same screen your code editor and a "terminal" that shows you successfully compiling a program. In case it isn't obvious already, this task is mostly free points—just do it and include a screen shot; we'll be happy.

Task 2: Diamond (2 points)

For this task, save your work in `Diamond.java`

Write a public class `Diamond`. Inside it, implement a function

```
public static void printDiamond(int k)
```

that takes a nonnegative integer `k` denoting the size of the diamond and prints to the screen (using `System.out`) a diamond of size `k`.

A diamond of size k contains a total of $2k-1$ lines, where every line contains a total of $2k+1$ characters (each character is either a `#` or a `*`). A few examples are given in Figure 1 for you to discover the pattern. Be warned that this task is **not** identical to a similar task you saw in Intro to Programming.

Specific Implementation Details: Your code can only use `for`-loops. You will print using either `print`, `println`, or `printf`. You may wish to learn about the `repeat` method, available for the `String` data type.

Task 3: Roman to Number (2 points)

For this task, save your work in `Roman.java`

You surely have encountered Roman numerals: I, II, III, XXII, MCMXLVI, etc. Roman numerals are represented by repeating and combing the following seven characters: `I` = 1, `V` = 5, `X` = 10, `L` = 50, `C` = 100, `D` = 500, `M` = 1000. To understand how they must be composed, we borrow the following excerpt from *Dive Into Python*:

- Characters are additive. `I` is 1, `II` is 2, and `III` is 3. `VI` is 6 ("5 and 1"), `VII` is 7, and `VIII` is 8.

- The tens characters (`I`, `X`, `C`, and `M`) can be repeated up to three times. At 4, you need to subtract from the next highest fives character. You can't represent 4 as `IIII`; instead, it is represented as `IV` ("1 less than 5"). The number 40 is written as `XL` (10 less than 50), 41 as `XLI`, 42 as `XLII`, 43 as `XLIII`, and then 44 as `XLIV` (10 less than 50, then 1 less than 5).
- Similarly, at 9, you need to subtract from the next highest tens character: 8 is `VIII`, but 9 is `IX` (1 less than 10), not `VIIII` (since the `I` character cannot be repeated four times). The number 90 is `XC`, 900 is `CM`.
- The fives characters cannot be repeated. The number 10 is always represented as X, never as `VV`. The number 100 is always `C`, never `LL`.
- Roman numerals are always written highest to lowest, and read left to right, so the order the of characters matters very much. `DC` is 600; `CD` is a completely different number (400, 100 less than 500). `CI` is 101; `IC` is *not* a valid Roman numeral (because you can't subtract 1 directly from 100; you would need to write it as `XCIX`, for 10 less than 100, then 1 less than 10).

Your Task: You will write a public class `Roman`. Inside it, implement a function

```
public static int romanToInt(String romanNum)
```

that takes a string that is a number represented using Roman numerals and returns the number, as an integer, equals in value to the input. For example:

- `romanToInt("I")==1`
- `romanToInt("V")==5`
- `romanToInt("VII")==7`
- `romanToInt("MCMLIV")==1954`
- `romanToInt("MCMXC")==1990`

Implementation Tips: Instead of a series of `if-else` constructs, you might wish to learn about the `switch`-case construct. This may simplify your life quite a bit.

For this problem, you don't need a `Map`, `HashMap`, or anything like that.

Task 4: N-Digit Fibonacci Number (2 points)

For this task, save your work in `Fib.java`

In this task, we'll play with the beloved Fibonacci sequence. The first two numbers in the sequence are 1 and 1. Each subsequent term is the sum of the previous two. Mathematically, we have that $F_1 = 1$, $F_2 = 1$, and for $n \geq 3$,

$$F_n = F_{n-1} + F_{n-2}.$$

Therefore, the initial 12 numbers of the sequence are: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144

As can be seen, the 12-th Fibonacci number, F_{12}, is 144, which has 3 digits.

Your Task: You will write a public class `Fib`, inside which you will implement a function

```
public static int firstNDigit(int n)
```

that takes an integer `n` as input and returns the smallest index k such that F_k has at least `n` digits. We'll only test your function with $k \leq 40,000$. For example:

- `firstNDigit(1)==1`
- `firstNDigit(2)==7`
- `firstNDigit(3)==12`

HINT: Consider using `BigInteger` (or `BigDecimal`) as F_{47} already exceeds the capacity of an `int` and F_{100} is easily beyond the capacity of a `long`.

Task 5: Subselection (2 points)

For this task, save your work in `Subsel.java`

This task involves writing a utility function for fixed arrays. You will make a public class called `Subsel`, inside which you will implement two functions

```
public static int[] takeEvery(int[] a, int stride, int beginWith) and
public static int[] takeEvery(int[] a, int stride)
```

with the following specifications:

- The call `takeEvery(a, s, b)` returns an `int` array containing the elements

  `a[b], a[b+s], a[b+2*s], a[b + 3*s],`....

  until the index in this progression falls out of `a`.
- The call `takeEvery(a, s)` has the same effect as calling `takeEvery(a, s, 0)`.
- The output may be empty; the `stride` parameter could be positive or negative—but never 0.

Abusing notation, this means:

- `takeEvery([1, 2, 3, 4], 2) == [1, 3]`
- `takeEvery([2, 7, 1, 8, 4, 5], 3, 2) == [1, 5]`
- `takeEvery([3, 1, 4, 5, 9, 2, 6, 5], -1, 5) == [2, 9, 5, 4, 1, 3]`

(*Hint #1:* We can have two functions with the same name; read about method overloading online.)
(*Hint #2:* One of them can be implemented by simply calling the other one.)

Task 6: Happy Time, Once Again (8 points)

For this task, save your work in `Happy.java`

You might remember happy numbers from a previous course. It is time for happy numbers once again. All your implementations for this task will live inside a public class called `Happy`.

Recap: Happy numbers are a nice mathematical concept. Just as only certain numbers are prime, only certain numbers are happy—under the mathematical definition of happiness. We'll tell you exactly what happiness means under this definition.

To test whether a number is happy, we can follow a simple step-by-step procedure:

(1) Write that number down

(2) Stop if that number is either 1 or 4.

(3) Cross out the number you have now. Write down instead the sum of the squares of its digits.

(4) Repeat Step (2)

When you stop, if the number you have is 1, the initial number is *happy*. If the number you have is 4, the initial number is *sad*. There are only two possible outcomes, happy or sad.

By this definition, 19 is a happy number because $1^2 + 9^2 = 82$, then $8^2 + 2^2 = 68$, then $6^2 + 8^2 = 100$, and then $1^2 + 0^2 + 0^2 = 1$. However, 145 is sad because $1^2 + 4^2 + 5^2 = 42$, $4^2 + 2^2 = 20$, and $2^2 + 0^2 = 4$.

Subtask I: First, you'll implement a function `public static long sumOfDigitsSquared(long n)` that takes a positive number `n` and returns the sum of each of its digits squared. For example,

- `sumOfDigitsSquared(7)` should return `49`
- `sumOfDigitsSquared(145)` should return `42` (i.e., $1^2 + 4^2 + 5^2 = 1 + 16 + 25$)
- `sumOfDigitsSquared(199)` should return `163` (i.e., $1^+9^2 + 9^2 = 1 + 81 + 81$)

Subtask II: Write a function `public static boolean isHappy(long n)` that takes as input a positive number `n` and tests if `n` is happy. You may wish to use what you wrote in the previous subtask.

- `isHappy(100)` should return `true`
- `isHappy(111)` should return `false`
- `isHappy(1234)` should return `false`
- `isHappy(989)` should return `true`

We will test your `isHappy` function with n up to $1,000,000$. Recursion (i.e., writing `isHappy` in a way that calls itself) may not fare well with large n.

Subtask III: Write a function `public static long[] firstK(int k)` that takes in a number $k \geq 0$ and returns an array of `long` of length exactly k that contains the first k happy numbers. For example: `firstK(11)` should return the array `{1, 7, 10, 13, 19, 23, 28, 31, 32, 44, 49}`.

We will only test this function with $k \leq 20,000,000$.

Task 7: Hidden String (2 points)

For this task, save your work in `Hidden.java`

Meow is playing a game with her friend. She gives her friend a string `s` and asks her whether another string `t` "hides" inside `s`. Let's be a bit more precise about hiding. Say you have two strings `s` and `t`. The string `t` hides inside `s` if all the letters of `t` appear in `s` in the order that they originally appear in `t`. There may be additional letters between the letters of `s`, interleaving with them.

Under this definition, every string hides inside itself: `"cat"` hides inside `"cat"`. For a more interesting example, the string `"cat"` hides inside the string `"afciurfdasctxz"` as the diagram below highlight the letters of `"cat"`:

`afciurfdasctxz.`

Importantly, these letters must appear in the order they originally appear. Hence, this means that `"cat"` does not hide inside the string `"xaytpc"`. In this case, although the letters `c`, `a`, and `t` individually appear, they don't appear in the correct order.

As another example, the string `"moo"` does not hide inside `"mow"`. This is because although we can find an `m` and an `o`, the second `o` doesn't appear in `"mow"`.

In this problem, you'll help Meow's friend by implementing a function

```
public static boolean isHidden(String s, String t)
```

that determines whether `t` hides inside `s`. The function returns a Boolean value: `true` if `t` hides inside `s` and `false` otherwise.

Here are some more examples:

```
isHidden("welcometothehotelcalifornia","melon") == true;
isHidden("welcometothehotelcalifornia","space") == false;
isHidden("TQ89MnQU3IC7t6","MUIC") == true;
isHidden("VhHTdipc07","htc") == false;
isHidden("VhHTdipc07","hTc") == true;
```