Before you begin. Read through this page of instructions. Do not start the exam (or access the next page) until instructed to do so.

Duration. You have 80 minutes to complete this exam.

Advice. Review the entire exam before starting to write code. Implement the constructor and methods in the order given, one at a time, testing in **`main()`** after you complete each method.

Submission. Submit your solutions on TigerFile using the link from the Exams page. You may submit multiple times (but only the last version will be graded).

Check Submitted Files. You may click the ***Check Submitted Files*** button to receive partial feedback on your submission. We will attempt to provide this feature during the exam, but you should not rely on it.

Grading. Your program will be graded *primarily* on correctness. However, efficiency and clarity will also be considered. You will receive partial credit for a program that implements some of the required functionality. You will receive a substantial penalty for a program that does not compile.

Allowed resources. This exam is open-book but not open-internet. During the exam you may use only the following resources: course textbook; companion booksite; lecture slides; course website; your course notes; your code from the programming assignments or precept; course Ed; course codePost, Java visualizer, and Oracle Javadoc. Accessing other websites or resources is prohibited. For example, you may not use Google, Google Docs, StackOverflow, or ChatGPT (or any other GenAI platform/assistant).

No collaboration or communication. During the exam, collaboration and communication (including sharing files) are prohibited, except with course staff members. A staff member will be outside the exam room to answer clarification questions.

No electronic devices or software. Software and computational/communication devices are prohibited, except to the extent needed for taking this exam (such as a laptop, browser, and IntelliJ). For example, you must close all unnecessary virtual desktops, applications, and browser tabs; disable notifications; and power off all other devices (such as cell phones, tablets, smart watches, and earbuds). *You must use only the Princeton wireless network eduroam, not a VPN, mobile hotspot or other network.*

Honor Code pledge. Write and sign the Honor Code pledge by typing the text below in the file **`acknowledgments.txt`**. Submit to TigerFile.

I pledge my honor that I will not violate the Honor Code during this examination.
Electronically sign it by typing **`/s/`** followed by your name.

After the exam. Discussing or communicating the contents of this exam before solutions have been posted is a violation of the Honor Code.

Overview. You are a bioinformatics engineer developing a toolkit for analyzing DNA sequences. Your task is to implement an **immutable** data type **`DNA`** that represents a DNA molecule and provides methods for common operations, and a client **`Translate`** that uses your class to produce protein sequences.

Deliverables. Submit the **`acknowledgments.txt`** file and two Java files:

1. **`acknowledgments.txt`** : contains your signed Honor Code pledge.
2. **`DNA.java`** : implements an *immutable* data type representing DNA sequences.
3. **`Translate.java`** : a client program that reads DNA sequences and translates them to protein sequences.

Restrictions: You may only use classes defined in **`java.lang`** or in our textbook libraries (such as **`StdOut`**, **`StdIn`**, **`In`**, **`Stack`**, **`Queue`**, **`ST`**, **`etc`**.). You may not use classes defined in **`java.util`**. Your program must not print anything beyond what is explicitly specified in the exam instructions. Your program must not throw any exceptions other than those explicitly required by the exam specifications.

Disclaimer: You do not need to know anything about biology or genomics to complete this exam. The DNA examples are used only to provide interesting and important context. Focus on the programming tasks, i.e., manipulating and analyzing sequences of letters according to the rules described below. All necessary background information is provided in this exam.

Part I (41 points): `DNA.java`. Using the template file `DNA.java` provided in the project folder as a starting point, write an immutable data type that implements the following API:

`public class DNA`	
`public DNA(String name, String sequence)`	creates a DNA object with the given name and sequence
`public int length()`	returns total number of nucleotides in the sequence
`public String sequence()`	returns the DNA sequence string
`public String toString()`	returns the string representation (see details below) of a DNA object
`public boolean isGene()`	returns `true` if this DNA sequence encodes a valid gene (see details below)
`public int distanceTo(DNA other)`	returns the Hamming distance between this DNA and another DNA - assuming the sequences are of equal length (see details below)
`public static void main(String[] args)`	test client - tests all public methods in the class.

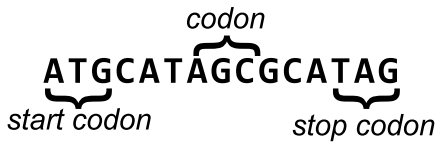
API details. Here is some additional information about the required behavior:

- The constructor. Each DNA sequence is a string that consists only of the four potential characters, called *nucleotides*: **`A`**, **`T`**, **`C`**, and **`G`**, represented as uppercase letters.

  Exception: Throw an `IllegalArgumentException` if the sequence contains invalid characters or if the name is an empty string. Lowercase letters are invalid characters.

- The **`toString()`** method returns a string representation of the format:
  `[name]: [sequence] ([length]nt)`
  For example, for a DNA sequence `ATGCCCTAGTAA` with the name *geneX*:
  `"geneX: ATGCCCTAGTAA (12nt)"`
  (here **`nt`** stands for nucleotide)

- The **`isGene()`** method tests whether the DNA sequence is a correct *protein-coding gene*, or simply a *gene*. The sequence is a correct gene if all of the following conditions are satisfied:
  1. The sequence can be split into consecutive *codons*, i.e., triplets of nucleotides.
  2. The sequence starts with one of the special *start codons*: either ATG or GTG.
  3. The sequence ends with one of the *stop codons*: either TAG, TAA, or TGA.
  4. When correctly split into consecutive codons, the sequence does not contain any intervening (or premature) stop codons. In other words, no stop codon is present in the sequence other than at the end of the sequence. For example:



  For example, the sequences `ATGCATAGCGCATAG` and `GTGCTGCTGTGA` are correct genes; the sequence `ATGCGCTGCGTCTGTACTAG` is not a correct gene because it has 20 nucleotides and therefore is not made up of codons; the sequence `ATGCCGTGACGTCTGTACTAG` is not a correct gene because it has a premature stop codon `TGA`.

- The **`distanceTo()`** method returns the *Hamming distance* between two sequences. The Hamming distance measures how many positions differ between two sequences of the same length. It is defined as the number of nucleotide positions at which the two sequences have different bases. For example:

```
        012345
DNA1:   ATGCTA
DNA2:   ATGATT
```

  Hamming distance = 2 (the sequences differ at positions 3 and 5, as shown above)

  Other examples: distance between `GATTACA` and `GCCTATA` is 3; distance between `CATCAT` and `GATGAT` is 2; distance between `TTAACCGG` and `TTAACCGG` is 0; distance between `CCGGAATT` and `CCGGAA` is not defined.

  Exception: Throw an `IllegalArgumentException` if sequences are of different length.

Part II (7 points): `Translate.java`. Using the template file `Translate.java` provided in the project folder as a starting point, write a client program Translate.java that takes the file name of a **codon translation table** as a command-line argument, reads the contents of that file, and stores the codon translation data in an appropriate data structure. The program then reads a gene name and its DNA sequence from standard input. If the DNA sequence represents a correct gene, the program uses the codon translation data structure to translate it and prints the gene name followed by the corresponding protein sequence, with one amino acid printed per line. Use your `DNA.java` implemented in Part I.

Translation is a mapping from a DNA sequence to *protein* sequence. A protein is a sequence of *amino acids*. There are 20 different amino acids. Each amino acid is represented by its 3-letter abbreviation, e.g., `Ala`, `Lys`, `Thr`, etc.

A DNA sequence that is a correct potential gene (as determined by the `isGene()` method) can be translated into a protein sequence:

- The DNA is split into consecutive codons (triplets of nucleotides).
- Each codon is mapped to a specific amino acid according to a codon translation table.
- *Stop codons* (`TAA`, `TAG`, `TGA`) signal the end of translation and are not translated into amino acids.
- There is a special translation rule for the alternative *start codon* `GTG` (see below).

The codon translation table is provided in a file named **`codons.txt`**. Each line of this file contains a codon and its corresponding amino acid (3-letter code), separated by a space. The file starts as follows:

```
AAA Lys
AAC Asn
AAG Lys
AAT Asn
ACA Thr
…
```

Note: In this table, stop codons are written with the symbol "*" so that the file has a uniform format. However stop codons do not correspond to amino acids and should not produce any output during translation. In other words, when translating a gene, stop codons should be treated as signaling the end of the protein and should contribute no amino acids to the result (see examples below).

Note: There is a special exception rule for an *alternative start codon* `GTG`: If a start codon is `GTG` (i.e., appears at the beginning of the gene), translate it to **`Met`**. Otherwise (i.e., if it is not a start codon), translate `GTG` to **`Val`** as specified in the codon translation table.

After compilation, your program should be run as follows:

```
java-introcs Translate codons.txt < gene.txt
```

where *`gene.txt`* is a file that begins with a gene name and DNA sequence separated by whitespace. (The file may contain other content afterwards for convenience; your program only needs to read the first two tokens: the name and the DNA sequence.)

For example, the file `geneA.txt` contains the gene name and DNA sequence:

```
geneA ATGTTTAAATGGGCTTAA
```

For the input gene, your program should:

1. Read the gene name and DNA sequence from standard input.
2. Create a DNA object using this name and sequence.
3. Check whether it is a correct gene (using the `isGene()` method).
4. If it is a correct gene, translate it into a protein and print:
    - the gene name on one line;
    - the protein sequence as 3-letter amino-acid codes, one per line.
5. Otherwise print:
    - the gene name on one line;
    - the word `Error` on the next line.

Here are some sample executions:

```
~/Desktop/f25-pe> java-introcs Translate codons.txt < geneA.txt
geneA
Met
Phe
Lys
Trp
Ala

~/Desktop/f25-pe> java-introcs Translate codons.txt < geneB.txt
geneB
Met
Arg
Leu
Val
Leu

~/Desktop/f25-pe> java-introcs Translate codons.txt < geneD.txt
geneD
Error
```

Your project folder contains several examples in the **.txt** files. Although these are not exhaustive text cases, you may use them to help you test your implementation of `Translate.java`. You are encouraged to write your own test cases.

We have also included some real-world examples for your own experimentation.