The purpose of this assignment is to give you practice using objects and data types. The first exercise focuses on the String ☕ data type; the second exercise involves the Color ☕ and Picture ▦ data types.

1. **Huntington's disease detector.** Huntington's disease is an inherited and fatal neurological disorder. Although there is currently no cure, in 1993 scientists discovered a very accurate genetic test. The gene that causes Huntington's disease is located on chromosome 4 and has a variable number of (consecutive) repeats of the CAG trinucleotide. The normal range of CAG repeats is between 10 and 35. Individuals with Huntington's disease have between 36 and 180 repeats. Doctors can use a PCR-based DNA test; count the maximum number of repeats; and use the following table to generate a diagnosis:

repeats	diagnosis
0–9	*not human*
10–35	*normal*
36–39	*high risk*
40–180	*Huntington's*
181–	*not human*

Write a program `Huntingtons.java` to analyze a DNA string for Huntington's disease and produce a diagnosis. To do so, implement the following public API:

```
public class Huntingtons {

    // Returns the maximum number of consecutive repeats of CAG in the DNA string.
    public static int maxRepeats(String dna)

    // Returns a copy of s, with all whitespace (spaces, tabs, and newlines) removed.
    public static String removeWhitespace(String s)

    // Returns one of these diagnoses corresponding to the maximum number of repeats:
    // "not human", "normal", "high risk", or "Huntington's".
    public static String diagnose(int maxRepeats)

    // Sample client (see below).
    public static void main(String[] args)

}
```

Here is some more information about the required behavior:

- *Maximum CAG repeats.* We recommend that you use the `substring()`, `equals()`, and `length()` methods from the `String` library.

- *Remove whitespace.* We recommend that you use the `replace()` method from the `String` library. Use `\n` to specify a newline character and `\t` to specify a tab character.

- *Performance requirement.* The `maxRepeats()` and `removeWhitespace()` methods must take time linear in the length of the string.

- *Sample client.* The `main()` method should

    - Take the name of a file as a command-line argument.

    - Read the genetic sequence from the file using the `In` class.

    - Remove any whitespace (spaces, tabs, and newlines).

    - Count the number of CAG repeats.

    - Print a medical diagnosis in the format below.

- *Input format.* The file format is a sequence of nucleotides (A, C, G, and T), with arbitrary amounts of whitespace separating the nucleotides. The data files dna4.txt, dna64.txt, chromosome4-hd.txt, and chromosome4-healthy.txt, are in the specified format.

Here are a few sample executions:

```
~/Desktop/oop1> cat repeats4.txt
TTTTTTTTTT TTTTTTTTTT TTTTTTTTCAGCAGCAGCAG TTTCAGCAGT TTTTTTTTTT
TTTTTTTTTT TTTTTTTTTT TTTTTTTTTTTTCAGTTTT TTTTTTTTTT T

~/Desktop/oop1> java-introcs Huntingtons repeats4.txt
max repeats = 4
not human

~/Desktop/oop1> cat repeats64.txt
TTTTTTTTTT TTTTTTTTTT TTTTTTTTTT TTTTTTTTTT TTTTTTTTTT TTCAGCAGCA
GCAGCAGCAG CAGCAGCAGC AGCAGCAGCA GCAGCAGCAG CAGCAGCAGC AGCAGCAGCA
GCAGCAGCAG CAGCAGCAGC AGCAGCAGCA GCAGCAGCAG CAGCAGCAGC AGCAGCAGCA
GCAGCAGCAG CAGCAGCAGC AGCAGCAGCA GCAGCAGCAG CAGCAGCAGC AGCAGCAGCA
GCAGTTTTTT TTTTTTTTTT TTTTTTTTTT TTTTTTTTTT TTTTTTTTTT TTTTTTTTTT

~/Desktop/oop1> java-introcs Huntingtons repeats64.txt
max repeats = 64
Huntington's

~/Desktop/oop1> java-introcs Huntingtons chromosome4-79.txt
max repeats = 79
Huntington's

~/Desktop/oop1> java-introcs Huntingtons chromosome4-19.txt
max repeats = 19
no Huntington's
```

2. **Kernel filter.** Write an image-processing library `KernelFilter.java` that applies various *kernel filters* to images, such as *Gaussian blur*, *sharpen*, *Laplacian*, *emboss*, and *motion blur*. A [kernel filter](#) modifies the pixels in an image by replacing each pixel with a linear combination of its neighboring pixels. The matrix that characterizes the linear combination is known as the *kernel*.
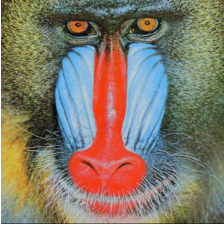
More specifically, to apply a kernel filter to a *grayscale* image, perform the following operation for each pixel p:

o Align the center of the kernel on pixel p.

o The new grayscale value of pixel p is obtained by multiplying each kernel element with the corresponding grayscale value, and adding the results.
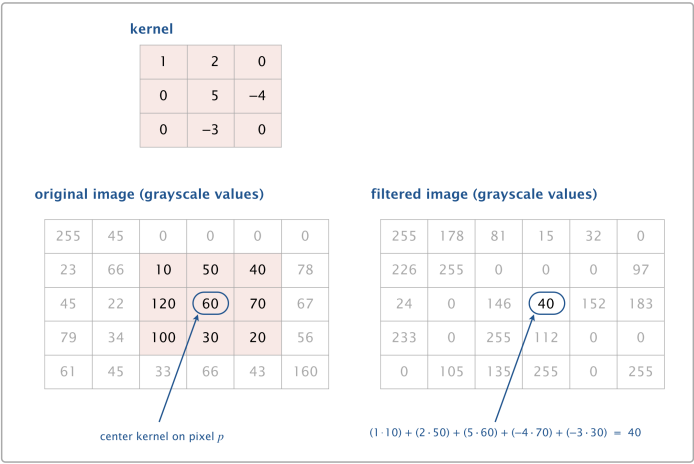
To apply a kernel filter to a *color* image, perform the above operation to the red, green, and blue components of each pixel p separately, and combine the results.
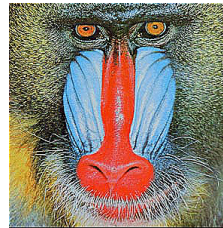
The following table describes the kernel filters that you will implement and illustrates the results using a classic test image ([baboon.png](#)):

image filter	kernel	result
identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Gaussian blur	$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

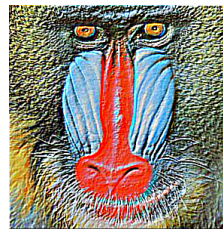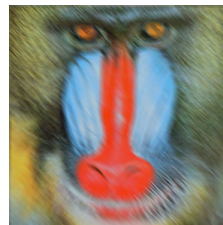|		
sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Laplacian	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
emboss	$\begin{bmatrix} -2 & -1 & 0 \\ -1 & 1 & 1 \\ 0 & 1 & 2 \end{bmatrix}$	
motion blur	$\frac{1}{9}\begin{bmatrix} 1&0&0&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0&0 \\ 0&0&0&0&1&0&0&0&0 \\ 0&0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&0&0&0&1 \end{bmatrix}$	

To do so, implement the following public API:

```
public class KernelFilter {

    // Returns a new picture that applies a Gaussian blur filter to the given picture.
    public static Picture gaussian(Picture picture)

    // Returns a new picture that applies a sharpen filter to the given picture.
    public static Picture sharpen(Picture picture)

    // Returns a new picture that applies an Laplacian filter to the given picture.
    public static Picture laplacian(Picture picture)

    // Returns a new picture that applies an emboss filter to the given picture.
    public static Picture emboss(Picture picture)

    // Returns a new picture that applies a motion blur filter to the given picture.
    public static Picture motionBlur(Picture picture)

    // Test client (ungraded).
    public static void main(String[] args)

}
```
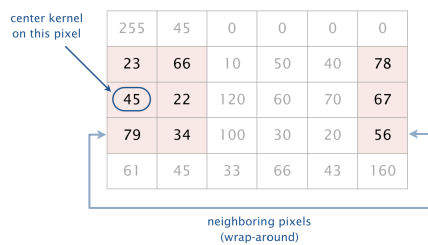
Here is some more information about the required behavior:

- *Periodic boundary conditions.* When applying a kernel filter to a pixel near the boundary, some of its neighboring pixels may not exist. In such cases, assume the leftmost column wraps around to the rightmost column and the top row wraps around to the bottom row (and vice versa).

center kernel on this pixel

255	45	0	0	0	0
23	66	10	50	40	78
45	22	120	60	70	67
79	34	100	30	20	56
61	45	33	66	43	160

neighboring pixels
(wrap-around)

- *Rounding.* When applying a kernel filter, the resulting RGB components may become fractional if the kernel weights are fractional. Round each RGB component to the nearest integer, with ties rounding up.

- *Clamping.* When applying a kernel filter, the resulting RGB components may not remain between 0 and 255. If an RGB component of a pixel is less than 0, set it to 0; if is greater than 255, set it to 255. This mechanism for handling arithmetic overflow and underflow is known as *clamping* or *saturating arithmetic*.

- *Test client.* Your `main()` method should test each of your public methods.

- *Performance requirement.* All methods should take time proportional to the product of the number of pixels in the image and the number of elements in the kernel.

Kernel filters are widely used for image processing in applications such as Photoshop or GIMP. They are also used in convolutional neural networks to extract features from an image and in digital cameras to remove camera shake.

Submission. Submit a `.zip` file containing `Huntingtons.java` and `KernelFilter.java`. Use only Java features that have already been introduced in the course (e.g., `String`, `In`, `Color`, and `Picture`, but not regular expressions).