6. Removing blanks from a string

Remove leading, trailing and double spaces from a given string.

This task often occurs when you need to clean the user input, such as from web forms, where leading or trailing spaces in, for example, names, are most likely user mistakes and should be removed.

Removing double and multiple spaces between words can be solved by using substitution:

```
my $string = 'Hello,     World!';
$string ~~ s:g/\s+/ /;
```

Don't forget to make the substitution global with the `:g` adverb to find all the occurrences of repeated spaces.

Leading and trailing spaces can be removed with the `trim` routine:

```
my $string = ' Hello, World! ';
say trim($string);
```

The `trim` routine exists as a self-standing function, as shown in the previous example, as well as a method of the `Str` class, so it can be called on a variable or on a string:

```
say $string.trim;
say ' Hello, World! '.trim;
```

There are also two routines, `trim-leading` and `trim-trailing`, which remove either only leading or only trailing spaces.

```
say '¡' ~ '  Hi  '.trim-leading;  # ¡Hi␣␣
say '  Hi  '.trim-trailing ~ '!'; # ␣␣Hi!
```

7. Camel case

Create a camel-case identifier from a given phrase.

It is a good practice to follow some pattern when choosing names for variables, functions, and classes in any programming language. In Raku, identifiers are case-sensitive, and, unlike many other languages, hyphens are allowed. So, variables names like `$max-span` or function names like `celsius-to-fahrenheit` are accepted.

In this task, we will form the `CamelCase` variable names from a given phrase. Names created in this style are built of several words; each of which starts with a capital letter.

Here's the program that does the required conversions:

```
my $text = prompt('Enter short text > ');
my $CamelName = $text.comb(/\w+/).map({.tclc}).join('');
say $CamelName;
```

All the actions are done in a sequence of method calls. The words are selected from the input `$text` using the `comb` method with a regex `/\w+/`. Then, each found word is `map`ped using the `tclc` method, which is equivalent to the chained call `.lc.tc`:

A 'bare' dot means that the method is called on the default variable `$_`, which is repeatedly set to the current element. In Raku, there is no `ucfirst` method to make the first letter of the text uppercase. Instead, we are using the `tclc` method; *tc* stands for *Title Case*, *lc* for *Lower Case*, and all the letters are converted to lowercase except the first one.

Finally, the elements of the array are joined together with the help of the `join` method. The input string 'Hello, World!' becomes `HelloWord` after all the transformations are done.

8. Incrementing filenames

Generate a list of filenames like file1.txt, file2.txt, etc.

Raku allows incrementing those kinds of filenames directly:

```
my $filename = 'file0.txt';
for 1..5 {
    $filename++;
    say $filename;
}
```

This program prints the list of consequent filenames:

```
file1.txt
file2.txt
file3.txt
file4.txt
file5.txt
```

Notice that after reaching 9, the *e* letter from *file* is incremented. Thus, `file9.txt` is followed by `filf0.txt`. To prevent that, add enough zeros in the template:

```
my $filename = 'file000.txt';
for 1..500 {
    $filename++;
    say $filename;
}
```

Now, the sequence starts with `file001.txt` and continues to `file500.txt`.

Multiple file extensions in the template, say `file000.tar.gz`, are also handled properly, so the numeric part is incremented.

9. Random passwords

Generate a random string that can be used as a password.

One of the possible solutions looks like this:

```
say ('0' .. 'z').pick(15).join('');
```

The `pick` method with no arguments takes a random element from the range of the ASCII characters between `0` and `z`. In the above example, calling `pick(15)` selects 15 different characters, which are then joined together using the `join` method.

It is important to know the two key features of the `pick` method. First, when called with an integer argument bigger than 1, the result contains only unique elements. Thus, all the characters in the password are different.

The second feature is a consequence of the first one. If the provided list of elements is not long enough, and its length is less than the argument of `pick`, the result is as long as the original data list.

To see which elements are used for generating a password, run the code with a number bigger than the whole ASCII sequence:

```
say ('0' .. 'z').pick(1000).sort.join('');
```

With this request, you will see all the characters that participate in forming a random password:

```
0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`
abcdefghijklmnopqrstuvwxyz
```

An example of a generated password: `05<EV]^bdfhnpyz`.