Exercises related to JavaScript programming

This document contains JavaScript / Node.js exercises to learn the basics of programming.

Kari Laitinen
http://www.naturalprogramming.com
2017-01-03 File created.
2018-10-29 Last modification

EXERCISES WITH PROGRAM Game.js

Exercise 1:

Now the program presents a number that is only minimally larger than the number given by the user of the program. Modify the program so that it outputs a number that is twice as large as the number typed by the user.

Exercise 2:

Improve the program further so that it prints three numbers to the user. The first number is the twice-as-large number that is calculated in the previous exercise. The other two numbers are numbers that follow the twice-as-large number. For example, if the user types in the number 144, your program must print numbers 288, 289, and 290. You should also modify the texts printed by the program. If the user types in the value 17, the output of the program should look like

```
This program is a computer game. Please,
type in a number : 17

You typed in 17.
My numbers are 34, 35, and 36.
Sorry, you lost the game.
I have more and larger numbers.
```

Exercise 3:

With the statement like

```
process.stdout.write( "\n The value of Pi is " + Math.PI + "\n" ) ;
```

you can print the value of the mathematical constant Pi to the screen. Improve the program so that it prints, after the previous printings, how much is the original given number divided by Pi. The character / is the division operator which you will need in this task. If the user typed in 17, a line like the following should be printed.

```
Your number divided by Pi is 5.411268065124442
```

Exercise 4:

The following statement prints the the square root of 2 to the screen:

```
process.stdout.write( "\n The square root of 2 is " + Math.sqrt( 2 ) + "\n" ) ;
```

Improve your program so that it prints the square root of the given number. If the user originally typed 17, a line like the following should be printed.

```
The square root of your number is 4.123105625617661
```

Exercise 5:

The program may not work well if the user types in a negative number. The value of the given number can be tested with an `if` construct like the following.

```
if ( given_number < 0 )
{
   process.stdout.write( "\n Negative numbers not allowed! \n\n" ) ;
   process.exit() ;
}
```

Try to figure out where into the program you should put the above `if` construct in order to handle the situation when the user types in a negative number.

EXERCISES RELATED TO CALCULATIONS AND DECISIONS

There are different units to express lengths and heights in the world. For example, in the U.S. it is common to use feet and inches to express human heights, whereas in continental Europe meters and centimeters are in use. These units relate to each other so that 1 foot is 30.48 centimeters, 1 inch is 2.54 centimeters, and 1 foot is 12 inches.

Exercise 1:

Write a program with which you can convert a human height given in centimeters to feet and inches. The program could produce the following output when executed:

```
This program can convert your height. Please,
type in your height in centimeters : 169

Your height in inches is 66.53543307086615
This is 5 feet and 6 inches.
```

To get the number of full feet, you need to be able to round downwards numerical values. This is possible when you use the mathematical function Math.floor(), which can be used in the following way.

```
var some_inches = 34.56 ;
var rounded_inches = Math.floor( some_inches ) ;
process.stdout.write( "\n Rounded value is " + rounded_inches ) ;
```

The above statements would print value 34 to the screen.

To start making this program, you can take, for example, the **Game.js** program and rename it to **Height.js**.

You could first convert the given centimeters to inches, and after that you can start thinking how to show the height in feet and inches.

Exercise 2:

Improve your program so that it prints additional information related to the given height. If the given height is less than 60 centimeters, the program must notice it in the following way.

```
This program can convert your height. Please,
type in your height in centimeters : 50

Your height in inches is 19.68503937007874
This is 1 feet and 7 inches.

Only 50 centimeters!
Are you really so short?
```

In this exercise you must write an `if` construct that prints the additional text lines if the given height is less than 60 centimeters.

Exercise 3:

Improve the `if` construct of your program so that the program says "That is a quite average height." if the given height is between 160 and 190 centimeters.

The program must also say something like "You might be a giraffe." if the given height is more than 400 centimeters.

Exercise 4:

If you have time, search the Internet and find out how to calculate an ideal weight for a person who has a certain height. Make your program to print the user’s ideal weight.

EXERCISES WITH PROGRAM Evenodd.js

These exercises are also related to making decisions in programs. To do decisions in programs, you must write `if-else` constructs.

Exercise 1:

In JavaScript there is a concept named NaN which means 'not a number'. For example, if you run the **Evenodd.js** program so that you give it the letter X when it asks a number, the program answers

```
NaN is odd.
```

The reason for this behavior is the fact that the text 'X' cannot be converted to a number, and thus the variable `given_number` gets the value `NaN`.

JavaScript has a global function named `isNaN()` which can be used to find out if a variable contains a NaN. This function can be used, for example, in the following way

```
if ( isNaN( given_number ) )
{
   process.stdout.write( "\n  You did not type in a number.\n\n" )
}
```

Now your task is to modify the **Evenodd.js** program so that you check with the `isNaN()` function whether the user typed in a number or something else.

You can put the existing `if-else` construct inside a new `else` block, in the following way:

```
if ( isNaN( given_number ) )
{
   process.stdout.write( "\n  You did not type in a number.\n\n" )
}
else
{
   // Here you should put the existing if-else construct.

   if ( ( given_number  %  2 )  ==  0 )
   {
      ...
}
```

Exercise 2:

Improve the program so that it checks whether the given number is in range 10 ... 1000, i.e., greater than 9 and less than 1001. If the given number does not belong to the said range, the program must inform the user about it, for example, in the following way

```
Please, give a number: 8

8 is even.

8 is not acceptable.
```

Exercise 3:

Improve the program so that the program generates a random number which will be compared to the number given by the user.

The random number must be close to the number given by the user, so that if the user gives the number 400, the random number must be in range 395 ... 405, or if the number given by the user is 850, the random number is in range 845 ... 855

If the random number generated by the program is greater than the number given by the user, the program must say that it has won the game. The program must also tell if it loses the game, or if the numbers are the same.

By studying the following program, you can find out how to generate random numbers.
http://www.naturalprogramming.com/jsprograms/nodejsfiles3/MathDemo.js

After this modification the program becomes a game, which is not always won by the computer. You must test the program so that all possible game results will be seen. About every 10th game should be a draw, i.e., both 'players' have the same number. Here is a possible output when the program is executed:

```
Please, give a number: 200

200 is even.

200 is in range 10 ... 1000

My number is 196. You won this game.
```

EXERCISES RELATED TO LOOPS

Exercise 1:

Write a program that prints a conversion table from miles to kilometers. The program should produce the following output to the screen

```
     miles     kilometers

     10.00       16.09
     20.00       32.19
     30.00       48.28
     40.00       64.37
     50.00       80.47
     60.00       96.56
     70.00      112.65
     80.00      128.74
     90.00      144.84
    100.00      160.93
    110.00      177.02
```

You can make this program by first making a copy of program **Miles.js** that we have studied earlier. To make **Miles.js** work locally on your computer, you must copy the file

```
http://www.naturalprogramming.com/jsprograms/nodejsfiles2/externals/tools.js
```

and place it in a subfolder named **externals** on your computer. Ensure that **Miles.js** works before you start doing the exercise.

You can first modify the program so that it produces the above output regardless of what the user types in from the keyboard.

You should use a `while` loop or a `for` loop in your program. If you use a `while` loop, the structure of the loop can look like the following. (In place of the four dots .... you need to put your own statements or expressions. Remember that it is good to indent the statements inside the braces { and }. Indentation means that you write the statements three character positions to the right.)

```
var  distance_in_miles  =  10 ;

var  distance_in_kilometers ;

while (  ....  )
{
   distance_in_kilometers = ....

   tools.printf( "\n %10.2f  %10.2f",
                 distance_in_miles, distance_in_kilometers ) ;

   distance_in_miles  =  distance_in_miles  +  10 ;
}
```

Exercise 2:

Improve the program so that it prints, after the table created in the previous exercise, a table that contains conversions from kilometers to miles. The table could look like the following.

```
kilometers     miles

    10.00        6.21
    20.00       12.43
    30.00       18.64
    40.00       24.86
    50.00       31.07
    60.00       37.28
    70.00       43.50
    80.00       49.71
    90.00       55.92
   100.00       62.14
   110.00       68.35
```

The above output can be generated with another loop. After this exercise is completed, your program should print two conversion tables.

Exercise 3:

Improve your program so that the user can select what kind of conversion table must be printed. In the beginning your program should print the following text.

```
This program prints conversion tables.
Type a letter to select a conversion table

 m   miles to kilometers
 k   kilometers to miles
```

After these lines are printed your program should read one character from the keyboard. According to the character the program should print the correct conversion table. You can detect the given character with the following line

```
var user_selection = String( input_from_user ).charAt( 0 ) ;
```

In this exercise you should add an `if-else if-else` construct to your program, and you must put the loops that were written in previous exercises inside the blocks of the `if-else if-else` construct. You should study program **Likejava.js** to find out how to organize the new version of your program. The `if` construct can begin in the following way.

```
if ( user_selection == 'm' || user_selection == 'M' )
{
   ...
```

Exercise 4:

Add a new selectable feature to your program. By pressing the letter P the user should be able to get a conversion table that contains conversions from pounds to kilograms. Pound is a unit of weight that is used in some countries. One pound is 0.4536 kilograms.

Exercise 5:

Improve the program so that it does not stop after it has printed a conversion table. Instead, it should have the exit from the program as a selectable feature. The program should print the following menu at the beginning as well as each time after a conversion table has been printed.

```
This program prints conversion tables.
Type a letter to select a conversion table

 m   miles to kilometers
 k   kilometers to miles
 p   pounds to kilograms
 x   exit the program
```

Inside the input function of the program, you should check if the user wants to exit the program, and only in that case you would use the statement

```
process.exit() ;
```

The input function of a JavaScript / Node.js program is such that it will always be executed on new input until the above statement is executed.

EXERCISES RELATED TO ARRAYS

Exercise 1:

Program **Reverse.js** stores numbers in an array and prints the given numbers in reverse order. Make a copy of **Reverse.js** and modify it so that it takes exactly seven numbers to the array. Also, the program must ensure that the given numbers are in the range from 1 to 40.

In this exercise you need to put a new `if` construct inside the input function of the program. You should not push the given number to the array if it does not belong to the specified range.

Make the program print the given numbers in 'normal' order instead of the reverse order.

Exercise 2:

Modify the program so that it will not accept a number if the number has already been typed in previously.

To do this, you need a loop that checks that the new given number is not among the numbers that have already been given to the program.

You have to put the new loop inside the input function in the program. You could also use a ’boolean’ variable to store information in the case that a given number has been entered

previously. You should put the new loop before the `if` construct that you put there in the previous exercise.

A ’boolean’ variable can be given two values: `true` or `false`. The necessary variable and the loop could look like:

```
var number_previously_given = false ;

// given_numbers.length now tells how many numbers have been given
// before the current number.

for ( let index_for_previous_numbers = 0 ;
          index_for_previous_numbers < given_numbers.length ;
          index_for_previous_numbers ++ )
{
   if ( given_numbers[ index_for_previous_numbers ] ==
                                          number_from_keyboard )
   {
      process.stdout.write( "\n   That number has already been given.\n" ) ;

      //  Here you have to modify the ’boolean’ variable.
   }
}
```

After the loop described above you should check in the `if` construct that the given integer is in the range 1 ... 40 and that the integer has not been entered previously. You can include the test of the value of the ’boolean’ variable into the boolean expression of the existing `if` construct.

Exercise 3:

After you have made the exercises above, you have a program that inputs numbers that could be used in Finnish national lottery game named Lotto.

Now you have to make the computer to generate its lottery numbers. In the folder
`http://www.naturalprogramming.com/jsprograms/nodejsfilesextra/`
you will find a program named **RandomNumbersInArray.js**. Copy suitable lines from that program to your program so that your program will have automatically generated lottery numbers.

After having done this, you have to improve your program so that it checks how many of the numbers given from the keyboard belong to the generated lottery numbers. You can think that the lottery numbers generated by the computer are the ’correct’ numbers, and by typing in numbers from the keyboard you try to ’play’ the lottery game.

To make testing of your program easier, you should make the program print the generated lottery numbers, so that you can enter some correct numbers from the keyboard.

Exercise 4:

Now the numbers used in the lottery game are in the range 1 ... 40. Try making this range smaller (e.g. 1 ... 14) and see how good results you can get by playing against the computer. In this exercise you should comment out some program lines so that it you cannot see the numbers generated by the computer.

EXERCISES RELATED TO STRINGS

Strings in computer programming are data structures that store textual information. A string usually contains a set of character codes which represent some particular text, such as a name entered from the keyboard, or a line of text from a file. In JavaScript, textual information is usually stored inside `String` objects.

Program **Widename.js** asks its user to type in a string from the keyboard, and after receiving the text it prints the characters separately as well as the hexadecimal character codes of the text. Make a copy of that program and do the following exercises.

Exercise 1:

Make the program print how many characters there are in the text that is entered from the keyboard. By studying program **StringReverse.js** you can see how the length of a string can be found out. After this modification a sample run of the program could look like

```
Please, type in your name: Kari Laitinen

Here is your name in a wider form:

  K a r i   L a i t i n e n

The characters as hexadecimal codes:

  4b 61 72 69 20 4c 61 69 74 69 6e 65 6e

The length of your name is : 13
```

Exercise 2:

In addition to the previous printings, make the program print the given name in reverse character order. Again, you can study **StringReverse.js** for help. The new output of the program could look like

```
Here is your name in reverse character order:

   nenitiaL iraK
```

Exercise 3:

Improve the program so that it prints your name also as Morse codes. By studying program **MorseCodes.js**, you can find out how this can be done. (If you do not know what Morse Codes are, stydy this article en.wikipedia.org/wiki/Morse_code)

As Morse code does not make a distinction between uppercase and lowercase letters, you should convert the characters of the given name to uppercase characters with a statement like

```
var uppercased_name = name_from_keyboard.toUpperCase() ;
```

The new output of your program could look like the following:

```
  Your name in Morse codes is:

   -.-   .-   .-.   ..          .-..   .-   ..   -   ..   -.   .   -.
```

Exercise 4:

Here your task is to improve the program so that it also prints the characters of your name in random order. One possibility to do this is to use a loop to remove the characters of your name randomly, and print them one by one until there are no characters left to print.

Because `String` objects cannot be modified or characters deleted from an existing string, you could convert the given name to an array of single-character strings and operate then with that `Array` object. It is possible to remove or delete the elements of an array.

The following program lines create an `Array` object from the `String` object, and remove a random element from the `Array`.

```
var characters_in_name = Array.from( name_from_keyboard ) ;

var random_character_index =
               Math.floor( Math.random() * characters_in_name.length ) ;

var removed_character = characters_in_name[ random_character_index ] ;

characters_in_name.splice( random_character_index, 1 ) ; // remove the character

process.stdout.write( removed_character ) ;
```

The `Array` property named `length` returns a value that tells how many characters are left in an `Array`. You could use the boolean expression `( characters_in_name.length > 0 )` to check when your `while` loop should terminate.

Exercise 5:

Improve the feature developed in the previous exercise so that the program produces five variations of the given name in random character order.

In this exercise you must put the loop created in the previous exercise inside a new loop. You can re-create the `Array` object inside the new loop.

The program could produce the following output if the given name is "Kari Laitinen"

```
Characters of your name in random order:

    airLiKnn atie
    ant LiiieaKnr
    rnniiaL iKeat
    irniaa LeKnti
    t KnieaiLnair
```


EXERCISES WITH PROGRAM Pyramids.js

You can find a program named **Pyramids.js** in the folder

`http://www.naturalprogramming.com/jsprograms/nodejsfilesextra/`

This program demonstrates the use of a function in a computer program. The program has a function named `print_pyramid()` that is called from the 'main' program. A numerical value should be given as a parameter for the `print_pyramid()` function. The numerical value specifies how many levels, or lines, the pyramid will have.

Make a copy of **Pyramids.js** to your own folder, and do the following exercises.

Exercise 1:

Modify the last lines of the program to find out how you can adjust the size of the pyramids. With the numerical value it is possible to specify the size, or height, of the printed pyramid.

Exercise 2:

Modify the definition of the function so that it begins in the following way:

```
function print_pyramid( desired_number_of_levels,
                        given_pyramid_character = '=' )
{
   ...
```


Modify also the body of the last `for` loop in the function so that the `given_pyramid_character` will be printed twice in the following way

```
process.stdout.write( given_pyramid_character ) ;
process.stdout.write( given_pyramid_character ) ;
```

After these modifications it will be possible to call the function so that you can specify in the function call the character with which the pyramid will be printed. For example, if you call the function in the following way

```
print_pyramid( 8, 'X' ) ;
```

the pyramid will be printed with letter 'X' in the following way

```
      XX
     XXXX
    XXXXXX
   XXXXXXXX
  XXXXXXXXXX
 XXXXXXXXXXXX
XXXXXXXXXXXXXX
XXXXXXXXXXXXXXXX
```

If the function is called without specifying the character, the default character '=' will be used.

Test that the function works correctly after these modifications.

Exercise 3:

Write a new function to the program so that it will be possible to print an inverted pyramid. If the new function is called

```
print_inverted_pyramid( 10 ) ;
```

a upside-down pyramid like the following should appear on the screen.

```
====================
 ==================
  ================
   ==============
    ============
     ==========
      ========
       ======
        ====
         ==
```

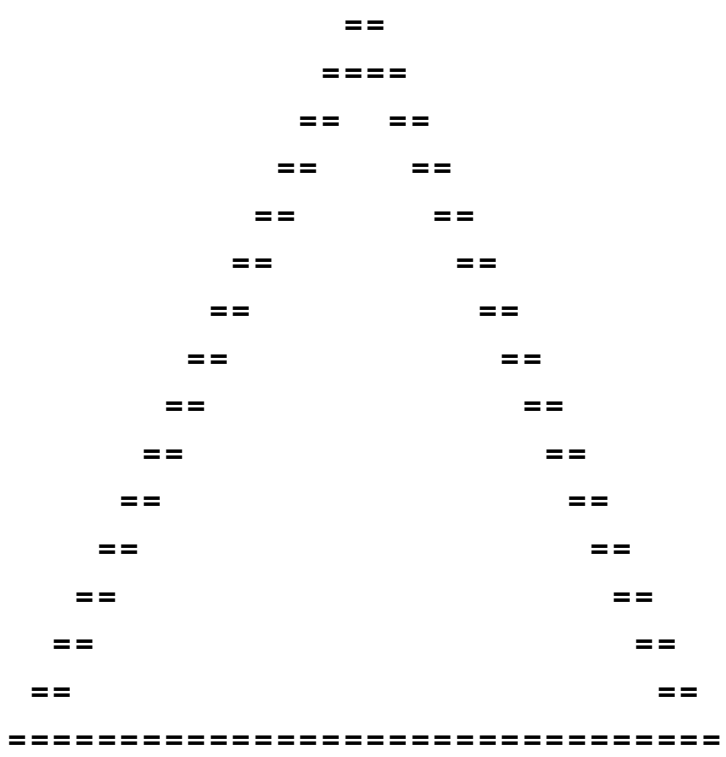
You can make the new function by first copying the function `print_pyramid()` and renaming it as `print_inverted_pyramid()`. In the new function, you should modify the header of the outer `for` loop so that the pyramid levels will be printed in opposite order when compared to the original function. You do not necessarily need to modify the body, i.e., the internal statements, of the `for` loop.

Exercise 4:

Write again a new function to the program so that if the new function is called like

```
print_hollow_pyramid( 16 ) ;
```

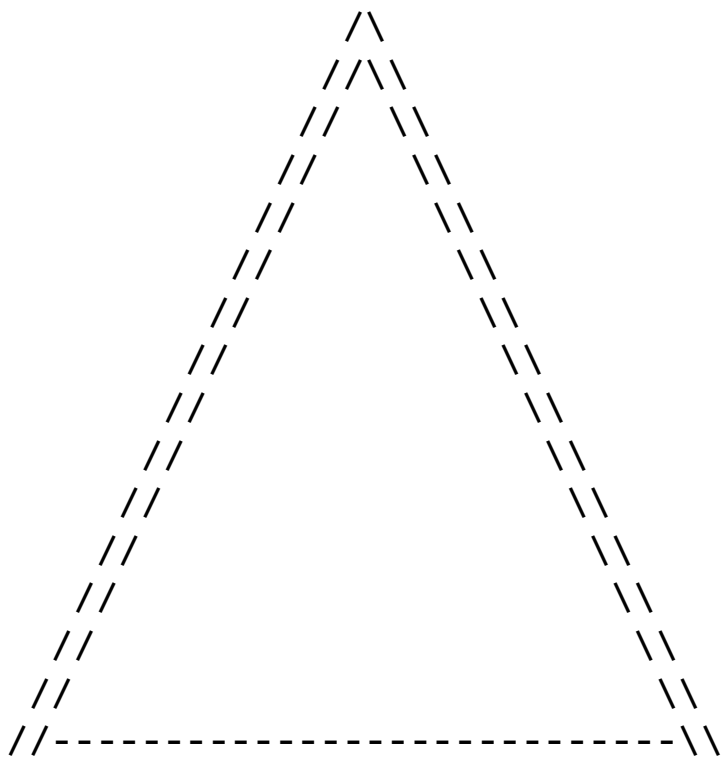
a hollow pyramid like the following will be printed



Also this new function can be made by first making a copy of the original function.

Exercise 5:

Improve the function that prints a hollow pyramid so that the walls of the pyramid are printed with character pairs "/\", "//" and "\\", in the following way



Note that when you want in your program to specify a string that contains two backslash characters, you must write it "\\\\" because backslash is a so-called escape character with which you can escape from the normal interpretation of characters. Then, two backslashes means that your really want a backslash printed.

EXERCISES RELATED TO FUNCTIONS WITH BigLetters.js

You can find a program named **BigLetters.js** in the folder
`http://www.naturalprogramming.com/jsprograms/nodejsfilesextra/`
This program demonstrates the use of a function in a computer program. The program has a function named `print_big_letter()` that is called from the 'main' program. A single-character string should be given as a parameter for the `print_big_letter()` function. The function then prints, if possible, the given character as a 'big letter' to the screen.

Make a copy of **BigLetters.js** to your own folder, and do the following exercises.

Exercise 1:

Improve the program so that it can print also the characters 'D' and 'E' as big letters.

For each printable letter the program has an array of strings that contains the strings that can be used to write the particular letter as a 'big letter'.

While doing this exercise it is good to learn to use Copy and Paste operations with your program editor.

Inside the `print_big_letter()` function, the letter data is printed with a `for-of` loop that processes all elements of an array. `for-of` loops are written with the keyword `for` but their structure differs slightly from the 'traditional' `for` loops. `for-of` loops can only be used with

arrays. All `for-of` loops can be replaced with traditional `for` loops. For example, the `for-of` loop

```
for ( let letter_data_line of letter_A_data )
{
   process.stdout.write( "\n " + letter_data_line ) ;
}
```

could be rewritten as the following traditional `for` loop

```
for ( let line_index = 0 ;
          line_index < letter_A_data.length ;
          line_index ++ )
{
   process.stdout.write( "\n " + letter_A_data[ line_index ] ) ;
}
```

As you can see, the `for-of` loop is shorter, and you do not have define an index variable for it. In the above `for-of` loop `letter_data_line` refers in turn to each string stored in the array referenced by `letter_A_data`.

Exercise 2:

Currently the `print_big_letter()` function contains a long `if ... else if ... else if ...` construct. That program structure is slightly complicated as all program blocks inside the long `if ... else if ... else if ...` contain a `for-of` loop. The structure of the program can be simplified if we write it as follows.

```
function get_letter_data( given_letter )
{
   var letter_data ;

   switch ( given_letter )
   {
      case 'A' : letter_data = letter_A_data ; break ;
      case 'B' : letter_data = letter_B_data ; break ;
      case 'C' : letter_data = letter_C_data ; break ;
      default:   letter_data = unknown_letter_data ;
   }

   return letter_data ;
}


function print_big_letter( given_letter )
{
   var letter_data = get_letter_data( given_letter )

   for ( let letter_data_line of letter_data )
   {
      process.stdout.write( "\n " + letter_data_line ) ;
   }

   process.stdout.write( "\n" ) ;
}
```

The above two functions could replace the original `print_big_letter()` function of the program. A new function named `get_letter_data()` is used to find and return the correct letter data. Function `get_letter_data()` uses a program construct named `switch-case` construct instead of the long `if ... else if ... else if ...` construct. `switch-case` constructs can be used instead of complicated `if ... else if ... else if ...` constructs in some cases. (By comparing programs **Likejavascript.js** and **LikejavascriptSwitch.js**, you can find more information about `switch-case` constructs.)

In this exercise your task is to modify the program so that you replace the original `print_big_letter()` function with the above two functions. You should also modify the `get_letter_data()` function so that the modifications made in the previous exercise still work.

It is very important that you try to understand how the above two methods work.

Exercise 3:

Write a new function named `print_big_wide_letter()` to the program. This new function should work so that, while the original `print_big_letter()` prints the character 'A' in the following way

```
   xx
  xxxx
 xx  xx
xx    xx
xxxxxxxx
xx    xx
xx    xx
```

the new `print_big_wide_letter()` should print the letter 'A' in the following way

```
      xxxx
    xxxxxxxx
  xxxx    xxxx
xxxx        xxxx
xxxxxxxxxxxxxxxx
xxxx        xxxx
xxxx        xxxx
```

The new function can use the same letter data as the original printing function. You can start making the new function by first making a copy of `print_big_letter()`. In the new function you must print each character of the letter data twice. You can use a for-of loop, a traditional `for` loop, or a `while` loop, to print each character of a line twice.

The following statement prints a character twice

```
        process.stdout.write( character_in_letter_data
                            + character_in_letter_data ) ;
```

To test the new function, you must call it from the ’main’ program.

Exercise 4:

Improve the program by adding yet another new function. The new function could be named `print_big_word()` and it could begin in the following way

```
function print_big_word( given_word )
{
      ...
```

If this new function is called from the ’main’ program in the following way

```
      print_big_word( "ABBA" ) ;
```

The following should appear on the computer screen

```
    XX     XXXXXXX   XXXXXXX      XX
   XXXX    XX    XX  XX    XX    XXXX
  XX  XX   XX    XX  XX    XX   XX  XX
 XX    XX  XXXXXXX   XXXXXXX   XX    XX
 XXXXXXXX  XX    XX  XX    XX  XXXXXXXX
 XX    XX  XX    XX  XX    XX  XX    XX
 XX    XX  XXXXXXX   XXXXXXX   XX    XX
```

Also in this exercise you should use the same letter data that is used by the other functions. Note that in the original letter data definitions there exist the necessary space characters at the end of each string in the letter data.

EXERCISES WITH PROGRAM GuessAWord.js

You can find a program named **GuessAWord.js** in the folder

`http://www.naturalprogramming.com/jsprograms/nodejsfilesextra/`

This program is a simple computer game in which the player has to try to guess the characters of a word that is 'known' by the game. Study the program and play the game in order to find out how the game has been programmed.

Exercise 1:

Improve the Guess-A-Word game so that the word to be guessed is randomly taken from an array of strings. Such an array can be created with a statement such as

```
var words_to_be_guessed =
        [ "VIENNA", "HELSINKI", "COPENHAGEN",
          "LONDON", "BERLIN", "AMSTERDAM" ] ;
```

A random index for an array such as the one above can be created with the `Math.random()` method with a statement like

```
var random_word_index =
      Math.floor( Math.random() * words_to_be_guessed.length ) ;
```

The `Math.random()` method returns a value in the range 0.0 .... 1.0 so that the value 1.0 is never returned. As the `Math.floor()` method is used, the calculated index is rounded 'downwards' to a suitable integer value.

Exercise 2:

Improve the program so that it counts how many guesses the player makes during a game. After a game is played, the program should print how many guesses were made. The following variable could be useful in this task

```
var number_of_guesses = 0 ;
```

Exercise 3:

Now the program is such that it terminates when the game is finished. Modify the program so that the game can be played several times during a single run of the program. In the above-mentioned folder there is a program named **RepeatableGame.js** which should be a helpful example.

Because you have to start a new game when the program starts running and when the user wants to play a new game, you could use a function that initializes the game. The initialization function should set suitable values to all global variables and arrays of the program.

Exercise 4:

Improve the program so that it prints game statistics before the program terminates. This means that the program shows which words were being guessed in played games, and how many guesses were made for each word. The game statistics could look like the following.

```
PLAYED WORD       GUESSES

COPENHAGEN          7
LONDON              6
COPENHAGEN          4
BERLIN              5
HELSINKI            4
```

As the ’played words’ will be randomly selected from an array, it is possible that the same word is played several times.

You can use the following kind of data items to store data of games:

```
var games_played = 0 ;
var played_words = [] ;
var guesses_in_games = [] ;
```

New data should be pushed to the end of the arrays after each game is played, and the data should be displayed on the screen in the end when the user no longer wants to play new games.

JavaScript Recap Exercises: A program to convert temperatures

There are two common systems for measuring temperature. Degrees of Fahrenheit (ºF) are used in the U.S. and some other countries, while degrees of Celsius (ºC) are in use in most European countries and in many countries throughout the world. The freezing point of water is 0 degrees Celsius and 32 degrees Fahrenheit, 10ºC is 50ºF, 20ºC is 68ºF, 30ºC is 86ºF, and so on. You can see that 10 degrees on the Celsius scale corresponds to 18 degrees on the Fahrenheit scale.

Exercise 1:

Write a program that asks the user to type in a temperature from the keyboard. After receiving the temperature value, the program must show how much that temperature is in both Degrees Celsius and in Degrees Fahrenheit. For example, if the user of the program types in 60, the output of your program should look something like

```
Please, give a temperature: 60

60.00 degrees Fahrenheit equals 15.56 degrees Celsius

60.00 degrees Celsius equals 140.00 degrees Fahrenheit
```

You can make this program by first making a copy of program **Miles.js** that we have studied earlier. By studying **Miles.js** you can find out how to do so-called formatted printing. To make **Miles.js** work locally on your computer, you must copy the file

```
http://www.naturalprogramming.com/jsprograms/nodejsfiles2/externals/tools.js
```

and place it in a subfolder named **externals** on your computer. Ensure that **Miles.js** works before you start doing the exercise.

Exercise 2:

Improve your program so that if the user types in the letter C, or lowercase c, the program prints a table that looks like

```
  Celsius          Fahrenheit

   0.00 C           32.00 F
  10.00 C           50.00 F
  20.00 C           68.00 F
  30.00 C           86.00 F
  40.00 C          104.00 F
  50.00 C          122.00 F
  60.00 C          140.00 F
  70.00 C          158.00 F
  80.00 C          176.00 F
  90.00 C          194.00 F
 100.00 C          212.00 F
```

You can make the table look good when you use the `tools.printf()` function that is used in **Miles.js**.

You need first to test with an `if` construct if the user types in a number or not-a-number. If the

input is not a number, then you must test if the user gave the letter ’C’ or ’c’. If the input is a number, then the program must behave as in Exercise 1. See the explanation in EXERCISES WITH PROGRAM Evenodd.js to find out how the `isNaN()` function can be used.

Exercise 3:

Improve your program so that if the user types in the letter F, or lowercase f, the program prints a table that looks something like

```
Fahrenheit        Celsius

  0.00 F      -     17.78 C
 10.00 F      -     12.22 C
 20.00 F      -      6.67 C
 30.00 F      -      1.11 C
 40.00 F             4.44 C
 50.00 F            10.00 C
 60.00 F            15.56 C
 70.00 F            21.11 C
 80.00 F            26.67 C
 90.00 F            32.22 C
100.00 F            37.78 C
```

Exercise 4:

Improve your program further so that the temperature tables are printed with separate functions in your program. You should call the functions when the user types in letters from the keyboard. Thus, your ’main’ program could have statements like

```
var user_selection = String( input_from_user ).charAt( 0 ) ;

if ( user_selection == 'F' || user_selection == 'f' )
{
   print_fahrenheit_to_celsius_table() ;
}
else if ( user_selection == 'C' || user_selection == 'c' )
{
   print_celsius_to_fahrenheit_table() ;
}
else
{
   process.stdout.write( "\n  Not valid selection." ) ;
}
```

This modification does not affect the behaviour of your program, but it improves the software structure.

EXERCISES WITH PROGRAM Animals.js

Animals.js is an example of object-oriented programming as it contains a class named `Animal`. Objects (instances) of type `Animal` are created with the `new` operator. Methods are called for the `Animal` objects with the dot operator.

Exercise 1:

Write a new method named `make_stomach_empty()` to class `Animal`. The new method could be called in the following way

```
dog_object.make_stomach_empty() ;
```

The new method should empty the 'stomach' of the `Animal` object so that an empty string "" is assigned to stomach contents. This same operation is done when `Animal` object is created, so you can copy a line from the constructor. You should test the new method by calling `make_speak()` after the stomach has been emptied.

Exercise 2:

Modify method `make_speak()` so that it prints something like

```
Hello, I am ...
My stomach is empty.
```

in the case when `stomach_contents` references just an empty string. The stomach is empty if method `feed()` has not been called for an `Animal` object, or if the stomach has been emptied

with the method that you wrote in the previous exercise. You can use the standard string property `length` to check if the stomach is empty. The `length` property can be used, for example, in the following way

```
if ( this.stomach_contents.length == 0 )
{
   // stomach_contents references an empty string.
   ...
```

If the stomach is not empty, the original output must be printed.

Exercise 3:

Modify the `feed()` method so that the animal complains if it has already eaten the given food. For example, if an `Animal` object is fed with the following two statements

```
dog_object.feed( "potatoes" ) ;
dog_object.feed( "potatoes" ) ;
```

the later call to the `feed()` method should produce a line like the following

```
I do not want more potatoes.
```

The `feed()` method must check if the `stomach_contents` string already contains the string that it receives as a parameter, and it must print a complaint if that is true. You must check if a string is contained in another string. The string method `indexOf()` can be used to do this:

```
if ( some_string.indexOf( another_string ) != -1 )
{
   // another_string is part of some_string
```

Exercise 4:

Rewrite the constructor of the `Animal` class in the following way:

```
constructor( given_parameter = "default animal",
             given_animal_name = "Nameless" )
{
   if ( typeof given_parameter == "string" ||
        given_parameter instanceof String )
   {
      this.species_name      =  given_parameter ;
      this.stomach_contents  =  "" ;
      this.animal_name       =  given_animal_name ;
   }
   else if ( given_parameter instanceof Animal )
   {
      this.species_name      =  given_parameter.species_name ;
      this.stomach_contents  =  given_parameter.stomach_contents ;
      this.animal_name       =  given_parameter.animal_name ;
   }
   else
   {
      process.stdout.write(
         "\n Unacceptable object was given to Animal constructor.\n" ) ;
   }
}
```

This new constructor adds a new data field `this.animal_name` to each `Animal` object. It

makes it possible to create an `Animal` object so that a name is given to the animal in the following way

```
var cat_object = new Animal( "cat", "Ludwig" ) ;
```

After you have rewritten the constructor, modify method `make_speak()` so that the new data field is printed, for example, in the following way.

```
Hello, I am a cat named Ludwig.
I have eaten: ...
```

The new constructor has default values for its parameters. With the new constructor it is possible to create an `Animal` object without giving any parameters, in the following way

```
var default_animal = new Animal() ;
```

Test what is printed to the screen when you call the `make_speak()` method for a 'default' `Animal`.

Exercise 5:

Modify program **Animals.js** so that you add there a new class named `Zoo`. You can write this new class after the `Animal` class. Objects of class `Zoo` should be objects that contain a set of `Animal` objects.

The `Zoo` class should have a method named `add_animal()` with which a new `Animal` object can be added to the zoo. Moreover, the `Zoo` class should contain a method named `make_animals_speak()`. Inside this method the `make_speak()` method should be called for

each `Animal` object. The `Zoo` class can look like the following:

```
class Zoo
{
   constructor()
   {
      this.animals_in_zoo = [] ;
   }

   add_animal( new_animal_to_zoo )
   {
      // push the new Animal object to the end of the array
   }

   make_animals_speak()
   {
      // call make_speak() for all Animal objects in a loop
   }
}
```

This `Zoo` class contains an array which stores `Animal` objects. By studying program **Olympics.js**, you can find out how an array of objects can be used.

You can test your new `Zoo` class with the following statements:

```
var test_zoo = new Zoo() ;

test_zoo.add_animal( cat_object ) ;
test_zoo.add_animal( dog_object ) ;
test_zoo.add_animal( another_cat ) ;

test_zoo.make_animals_speak() ;
```

Exercise 6:

If you have still time and enthusiasm left, write a method named `feed_all()` to the `Zoo` class. With this method it should be possible to feed all animals of the zoo with the same food string.

EXERCISES WITH PROGRAM Olympics.js

Exercise 1:

Program *Olympics.js* has an initialized array to store objects of type `Olympics`. Update the olympics list of the program so that the latest known Summer Olympic games have `Olympics` objects created. You can also add information about future Summer Olympics.

Ensure that the program works correctly after these modifications.

Exercise 2:

Create a new class named `WinterOlympics` so that the new `WinterOlympics` class will be a subclass of the original `Olympics` class. The first definition of the `WinterOlympics` class could be the following

```
class WinterOlympics extends Olympics
{
   constructor( given_olympic_year,
                given_olympic_city,
                given_olympic_country )
   {
      super( given_olympic_year,
             given_olympic_city,
             given_olympic_country ) ;
   }
}
```

You can put the new `WinterOlympics` class right after the `Olympics` class in the program. With the keyword `extends` we can make `WinterOlympics` to inherit the `Olympics` class. We can say also that class `WinterOlympics` is *derived* from the `Olympics` class. Class `Olympics` is is the superclass of class `WinterOlympics`. With the keyword `super` the constructor of superclass can be called.

When you write the `WinterOlympics` class as shown above, it will behave in the same way as its superclass `Olympics`.

You can test your new class by adding the following object to the array

```
new WinterOlympics( 2006, "Turin", "Italy" )
```

It is possible to have both `Olympics` and `WinterOlympics` objects in the same array. If your program can find the data of Turin olympics, you have successfully carried out this exercise.

Exercise 3:

Improve the new `WinterOlympics` class by writing a new version of method `get_full_data()` into it. The new method should return a text that contains the word 'winter'. The output of the method could look like the following

```
In 2006, Winter Olympics were held in Turin, Italy.
```

In this exercise you can copy the corresponding method from class `Olympics`, and modify the text that is generated by the method.

When a subclass contains a method that has the same name and similar parameters as a method in the superclass, we say that the method is *overridden* in the subclass. In this exercise we override the method `get_full_data()` and the new version of the method will automatically be used for `WinterOlympics` objects.

Exercise 4:

Earlier the winter and summer olympics were orginized during the same year. For example, in 1984 the winter olympics were in Sarajevo, Yogoslavia, and the summer olympics were in Los Angeles, U.S.A.

If you add `WinterOlympics` objects that describe these earlier winter games to the olympics list of the program, there will be problems. The search algorithm finds only the first olympics that were held in the given year. If there are two objects for the same year, the latter object will not be found.

Modify the program so that it will always process all objects of the list of olympics, and print data of those objects that contain the given year.

One way to do this is that you use another array to store those `Olympics` objects that need to be printed. Now the program has the variable

```
var selected_olympics = null ;
```

You can modify this statement so that you specify an empty array in the following way

```
var selected_olympics = [] ;
```

Then you can go through all `Olympics` objects of `olympics_list`, and with the `push()` method add those objects whose year matches the given year to the above array.

After all `Olympics` objects have been checked, you can test the `length` of the above array, and print information about as many `Olympics` as there are objects in the above array.

To test the new version of the program you should add more `WinterOlympics` objects to the initialized array.

Exercise 5:

Now the **Olympics.js** program accepts only numbers as input from the user. Improve the program so that if the user types in the word "summer", the program will print data of all summer olympic games. Then, if the user types "winter", the data of all winter games will be printed.

JavaScript has a built-in global function named `isNaN()` which returns `true` if something is 'not-a-number'. You can use this function to test if the user typed in a number or a word. The function can be called in the following way

```
if ( isNaN( input_from_user ) )
{
   var given_text = String( input_from_user ).trim() ;
   ...
```

As shown above, you should convert the user input to a string and `trim()` it before you start checking what word was typed.

Another you have to solve while doing this exercise is to find out how you decide whether an object in the array is of type `Olympics` or of type `WinterOlympics`. You can solve this problem by using the `instanceof` operator of JavaScript.

The boolean expression in the following `if` construct will be `true` if `olympics_object` refers to an object that is of type `WinterOlympics` or of some subtype of `WinterOlympics`.

```
if ( olympics_object instanceof WinterOlympics )
{
   ...
}
```

Then, if you would like to know whether some olympics object is of type `Olympics`, you can use an `if` construct such as

```
if ( ! ( olympics_object instanceof WinterOlympics ) )
{
   ...
}
```

In this exercise, you cannot use the `instanceof` operator to test whether an object is of type `Olympics`. The reason for this is that `instanceof` returns `true` also when the object on its left side is an object of some subclass of the class given on the right side of the operator.

Exercise 6:

If the user types in a text but the text is not "summer" or "winter", your program should suppose that the user typed in the name of a country or a city.

Make an `else` part to your program so that it searches all `Olympics` or `WinterOlympics` objects that contain the given text. For example, if the user types "U.S.A.", the program should print a list like the following:

```
In 1904, Olympic Games were held in St. Louis, U.S.A..
In 1932, Olympic Games were held in Los Angeles, U.S.A..
In 1984, Olympic Games were held in Los Angeles, U.S.A..
In 1996, Olympic Games were held in Atlanta, U.S.A..
```

Or if the user types "Italy", the program should print something like

```
In 1960, Olympic Games were held in Rome, Italy.
In 2006, Winter Olympics were held in Turin, Italy.
```

EXERCISES WITH PROGRAM CapitalsMap.js

CapitalsMap.js is a program that shows how to use a `Map` object in JavaScript. `Map` is a built-in standard class that can be used in all JavaScript programs. A `Map` can store information so that it is easy to find the stored information.

A `Map` can store key-value pairs, and when you know the ’key’ it is easy to get the correrponding ’value’. It is also possible to ask a `Map` if it has a ’value’ associated with a certain ’key’.

Keys and values stored in a `Map` can be objects or primitive data items.

By studying **CapitalsMap.js** you can find out that

- method `set()` can be used to put a new key-value pair to a `Map`
- with method `has()` you can check whether a ’key’ has a ’value’ in the `Map`
- method `get()` returns the ’value’ associated the the specified ’key’

Exercise 1:

Add new Country-Capital pairs to the `Map` in the program, and check that the program works with the new data.

Exercise 2:

Improve the program so that when it doesn't find the capital associated with the given country name, it prints a list of all countries whose capitals the program knows.

The program should thus work in the following way:

```
This program may know the capital of a country.
Type in the name of some country: Australia

Sorry, I do not know the capital of Australia.

I know only the capitals of the following countries:
 Afghanistan  Austria  Belgium  Chile  Denmark  England
 Finland  France  Holland  Hungary  Iceland  Israel
 Italy  Japan  Norway  Pakistan  Poland  Portugal
 Russia  Spain  Sweden  Usa
```

Here you should print all the 'keys' used in the `Map`.

You can get the keys with the following statement

```
var country_names = countries_and_capitals.keys() ;
```

and you can print the country names with a `for-of` loop in the following way

```
for ( let country_name of country_names )
{
   process.stdout.write( "  " + country_name ) ;
}
```

You can try the above solution first, but to completely do this exercise, you should print the country names in alphabetical order, and you should find out a way how you print no more than 6 country names on each line.

You can get a sorted array from the above country_names with the statement:

```
var sorted_country_names = Array.from( country_names ).sort() ;
```

By printing the country names of this sorted array, you get the names in alphabetical order. The data structure returned by the `keys()` method of `Map` is not a real array although it can be used in `for-of` loop like an array.

Exercise 3:

Add the following class to the beginning of the program

```
class City
{
   constructor( given_city_name,
                given_population,
                given_web_address )
   {
      this.city_name = given_city_name ;
      this.population = given_population ;
      this.web_address = given_web_address ;
   }

   get_city_name()
   {
      return this.city_name ;
   }

   print_city_info()
   {
      process.stdout.write( "\n " + this.city_name + " has population "
                          +  this.population + "\n More information at: "
                          +  this.web_address + "\n\n" ) ;
   }
}
```

When you have the `City` class in use, you can put key-value pairs to the `Map` in the following way

```
countries_and_capitals.set( "Sweden", new City( "Stockholm", 940000,
                                                "www.stockholm.se" ) ) ;
```

Your task is here now to modify the program so that it can handle the situation when the 'value' in the `Map` is a `City` object. This means that the program must call the `print_city_info()` method in the following way:

```
This program may know the capital of a country.
Type in the name of some country: Sweden

The capital of Sweden is Stockholm.

Stockholm has population 940000
More information at: www.stockholm.se
```

You can use the `instanceof` operator to find out if there is a `City` object as a 'value' in the `Map`.

```
var  capital_data  =  countries_and_capitals.get( country_name ) ;

if ( capital_data instanceof City )
{
```

If the 'value' is not a `City` object, the program can assume that it is a string.

Exercise 4:

This exercise does not continue the development made in the previous exercises. Therefore, make a copy of the current version of your program. Your task here is to modify the program so that it becomes a game in which the user must know the capital of a country or the country of a capital.

The name of the copied program could be **CapitalsGame.js**

When this program is executed it automatically selects either a capital or a country and presents it to the user.

If the program is showing a country name, the user should respond with the corresponding capital name.

If the program is showing a capital name the user should respond with the corresponding country name.

In earlier exercises you have learned how to make random selections in a program.

In the previous exercise you learned that with the `keys()` method you can get a list of keys. There is a corresponding method named `values()` with which you can get a list of values from the `Map`.

Note that you must be able to handle the case when a 'value' is a `City` object.