

Lecture 2: JavaScript Fundamentals

Written by Ben Yan, from materials by Jerry Cain

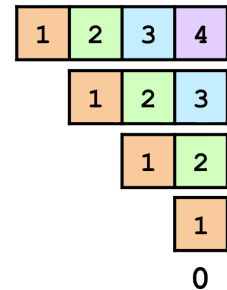
Digit Sum

One of the most useful and ubiquitous programming techniques is the **while** loop, which repeatedly runs a block of code as long as some condition remains true. When that condition is tragically broken (no longer true) at the end of an iteration, the loop is exited. In JavaScript, the **while** loop has structure

```
while ( condition ) {
    block of statements to be repeated
}
```

An example use case is a function that **computes the sum of a number's digits**.

```
function digitSum(n) {
    let sum = 0;
    while (n > 0) {
        sum += n % 10;
        n = Math.floor(n / 10);
    }
    return sum;
}
```



The **while** loop is leveraged to read its digits from right to left, one at a time. Given an input **n**, we extract the current one's digit via **n % 10**, then trim it by dividing by **10** and rounding down, e.g., **1234** to **123**, by **Math.floor(n / 10)**. This repeats until **n** has been divided down to **0**, i.e. all its digits have been trimmed away. Note that **sum** accumulates all the digits' values before we return it.

Factorial Table

An equally famous and important structure is the **for** loop, which specifies the lifespan or runtime of the loop independently from what operations the loop performs. The syntax in JavaScript appears as:

```
for ( init ; condition ; step ) {
    block of code to be repeated
}
```

Here's a step-by-step cookbook on how JavaScript evaluates a **for** loop.

1. Evaluates **init**, which typically declares a control variable, e.g., **let i = 0**.
2. Evaluates **condition** and exit from the loop if the value is false, e.g., we check that **i < 100**.
3. Executes the block of statements in the body of the loop.
4. Evaluates **step**, which usually updates the control variable, e.g., **i++** to increment by **1**.
5. Return to Step 2 to begin the next loop cycle or iteration.

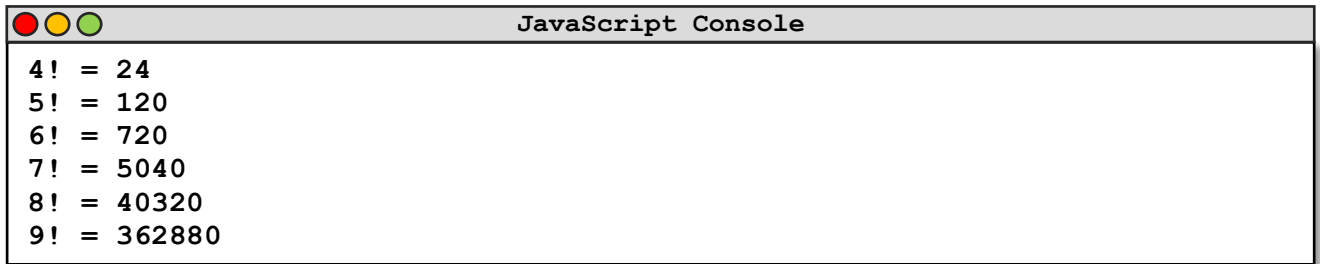
An interesting introductory example is a **for** loop that **yields the factorial of a number n** , computed by multiplying together all positive integers up to and including n , i.e. $n! = 1 * 2 * 3 * \dots * n$.

```
function factorial(n) {
  let result = 1;
  for (let i = 1; i <= n; i++) {
    result = result * i;
  }
  return result;
}
```

By repeatedly calling the helper function **factorial** inside another **for** loop, we can print out a table of the factorial values of all integers in a certain range, as circumscribed by bounds **min** and **max**.

```
function factorialTable(min, max) {
  for (let i = min; i <= max; i++) {
    console.log(i + "! = " + factorial(i));
  }
}
```

For instance, the call **factorialTable(4, 9)** would print out the neat summary:



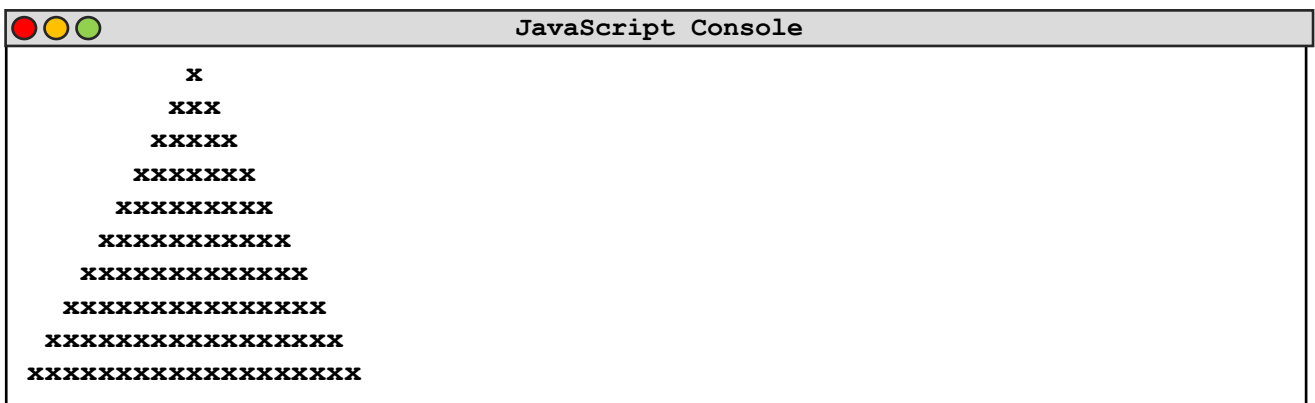
```
JavaScript Console
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
```

Console Pyramid 🧱

We'll implement a function **ConsolePyramid**, which displays a *Super Mario* esque pyramid on the console, with each brick rendered as the letter **"x"**. Assuming these constants have been defined:

```
const N_LEVELS = 10;
const BRICK_CHARACTER = "x";
```

Then a call to **ConsolePyramid()** would result in the following levels or lines being printed.



```
JavaScript Console
  x
 xxx
xxxxx
xxxxxxx
xxxxxxxxx
xxxxxxxxxxx
xxxxxxxxxxxxx
xxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxx
```

Notice with **10** levels, the 1st level is 9 spaces followed by 1 brick, the 2nd level is 8 spaces followed by 3 bricks, the 3rd level is 7 spaces followed by 5 bricks, and so on, with the 10th and final level being 0 spaces followed by 19 bricks. To generalize, **for a pyramid with `N_LEVELS`, the `i`th level consists of $(N_LEVELS - i)$ spaces followed by $(2 * i - 1)$ bricks, where $1 \leq i \leq N_LEVELS$.**

With this in mind, we'll write **`concatNCopies`**, which produces a string with **`N`** copies of the same letter, e.g., a space or brick. Note the fashionable use of a loop to build up the string one letter at a time.

```
function concatNCopies(n, str) {
  let result = "";
  for (let i = 0; i < n; i++) {
    result += str;
  }
  return result;
}
```

For instance, **`concatNCopies(3, " ")`** returns " ", which you can verify is indeed 3 empty spaces ☺, and **`concatNCopies(4, "X")`** returns "XXXX", which sort of looks like a swear word.

Now, we can write the main program, using a loop to print each level of the pyramid to console!

```
function ConsolePyramid() {
  for (let level = 1; level <= N_LEVELS; level++) {
    let spaces = concatNCopies(N_LEVELS - level, " ");
    let bricks = concatNCopies(2 * level - 1, BRICK_CHARACTER);
    console.log(spaces + bricks);
  }
}
```

Ruth Aaron Pairs

If you've been keeping up with this history of America's pastime, perhaps you're familiar with baseball greats **Babe Ruth** (New York Yankees) and **Hank Aaron** (Atlanta Braves). Ruth held the record for career home runs at 714, until Aaron hit his 715th on April 8th, 1974, against the Los Angeles Dodgers.

Incidentally, it turns out **714** and **715** have the curious mathematical property that the sum of each of their *distinct* prime factors is equal. For **714**, we have $2 + 3 + 7 + 17 = 29$, and for **715**, we have $5 + 11 + 13 = 29$. And so mathematicians, being mathematicians, came up with **Ruth-Aaron pairs**, which are two neighboring integers that satisfy the property above, e.g., also **5** and **6**, **24** and **25**, etc ☺.

We'll implement the function **`isRuthAaronNumber`**, which accepts an integer **`n`** as input, and returns true only if **`n`** and **`n + 1`** are Ruth-Aaron pairs, and false otherwise. Ultimately, we'll use this to discover and print out the first **15** Ruth-Aaron numbers. The starting interface of our program appears as follows.

```
const NUM_PAIRS = 15;

function RuthAaronPairs() {
  console.log("Here are the first " + NUM_PAIRS + " Ruth Aaron pairs:");
  console.log("");
  printRuthAaronPairs(); // a function that we'll implement below!
}
```

First, we'll need an algorithm that computes the sum of a number's **distinct prime factors**. The following is a clever approach that uses an outer loop to iterate through possible factors. Once a factor of **n** is discovered, i.e. **n % factor === 0**, an inner loop is used to divide out all copies of factor from **n**'s prime factorization. We keep going until all factors have been divided out, i.e. **n** dwindles down to **1**.

```
function sumOfDistinctPrimes(n) {
  let sum = 0;
  let factor = 2; // 1 is not considered a prime number
  while (n > 1) {
    if (n % factor === 0) {
      sum += factor;
      while (n % factor === 0) {
        n /= factor;
      }
      // while loop gets rid of all copies of factor, even
      // though only one contributes to sum
    }
    factor++;
  }
  return sum;
}
```

For instance, if **n = 72**, this function will find **2** is a factor, and divide **n** by **2** thrice, so **n -> 9**. Then, we'll find **3** is a factor, and divide twice so **n -> 1**. We thus have **2** and **3** as distinct prime factors, which sum to **5**. Note this inner division loop ensures **factor**, if it's added to the **sum**, is truly prime, as any composite factor, e.g., **4**, **6**, would have had its own prime factors divided out in an earlier cycle.

From here, a true / false function that checks if **n** and **n + 1** are Ruth Aaron pairs is not too painful.

```
function isRuthAaronNumber(n) {
  let first = n;
  let second = n + 1;
  return sumOfDistinctPrimes(first) === sumOfDistinctPrimes(second);
}
```

Lastly, we can write a **while** loop that keeps going until we've found **NUM_PAIRS** Ruth-Aaron pairs!

```
function printRuthAaronPairs() {
  let n = 1; // check every positive integer starting from 1
  let numFound = 0;
  while (numFound < NUM_PAIRS) {
    if (isRuthAaronNumber(n)) {
      numFound++;
      console.log("  + (" + n + ", " + (n + 1) + ")");
    }
    n++;
  }
}
```

Note the elegant decomposition of this task into smaller, readable, and descriptive helper functions ☺.