Jerry Cain and Eric Roberts
CS 106J

Handout #15
April 19, 2017

Mechanics of Functions

Mechanics of Functions

Jerry Cain and Eric Roberts
CS 106J
April 19, 2017

Mechanics of the Function-Calling Process

When you invoke a function, the following actions occur:

1. JavaScript evaluates the arguments in the context of the caller.
2. JavaScript copies each argument value into the corresponding parameter variable, which is allocated in a newly assigned region of memory called a ***stack frame***. This assignment follows the order in which the arguments appear: the first argument is copied into the first parameter variable, and so on. If there are too many arguments, the extras are ignored. If there are too few, the extra parameters are initialized to **`undefined`**.
3. JavaScript then evaluates the statements in the function body, using the new stack frame to look up the values of local variables.
4. When JavaScript encounters a **`return`** statement, it computes the return value and substitutes that value in place of the call.
5. JavaScript then removes the stack frame for the called function and returns to the caller, continuing from where it left off.

The Combinations Function

- To illustrate method calls, the text uses a function $C(n, k)$ that computes the ***combinations*** function, which is the number of ways one can select k elements from a set of n objects.
- Suppose, for example, that you have a set of five coins: a penny, a nickel, a dime, a quarter, and a dollar:

How many ways are there to select two coins?

penny + nickel
penny + dime
penny + quarter
penny + dollar

nickel + dime
nickel + quarter
nickel + dollar

dime + quarter
dime + dollar

quarter + dollar

for a total of 10 ways.

Combinations and Factorials

- Fortunately, mathematics provides an easier way to compute the combinations function than by counting all the ways. The value of the combinations function is given by the formula

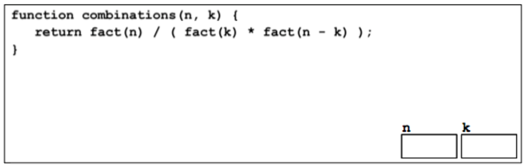
$$C(n,k) = \frac{n!}{k! \times (n-k)!}$$

- Given that you already have a **`fact`** function, is easy to turn this formula directly into a function, as follows:

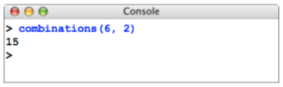
```
function combinations(n, k) {
   return fact(n) / (fact(k) * fact(n - k));
}
```

- The next slide simulates the operation of **`combinations`** and **`fact`** in the context of a simple **`run`** function.

Tracing the **`combinations`** Function



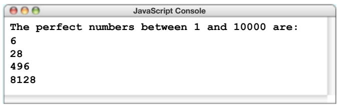
```
function combinations(n, k) {
   return fact(n) / ( fact(k) * fact(n - k) );
}
```



```
> combinations(6, 2)
15
>
```

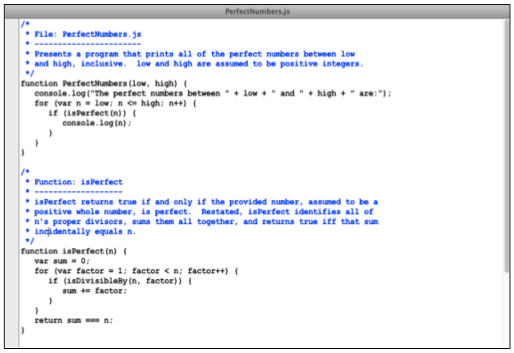
Exercise: Finding Perfect Numbers

- Greek mathematicians took a special interest in numbers that are equal to the sum of their proper divisors (a proper divisor of n is any divisor less than n itself). They called such numbers ***perfect numbers***. For example, 6 is a perfect number because it is the sum of 1, 2, and 3, which are the integers less than 6 that divide evenly into 6. Similarly, 28 is a perfect number because it is the sum of 1, 2, 4, 7, and 14.
- For the next several minutes of class, we're going to design and implement a JavaScript program that finds all the perfect numbers between two limits. For example, if the limits are 1 and 10000, the output should look like this:



```
The perfect numbers between 1 and 10000 are:
6
28
496
8128
```

PerfectNumbers.js



```
/*
 * File: PerfectNumbers.js
 * ----------------------
 * Presents a program that prints all of the perfect numbers between low
 * and high, inclusive.  low and high are assumed to be positive integers.
 */
function PerfectNumbers(low, high) {
   console.log("The perfect numbers between " + low + " and " + high + " are:");
   for (var n = low; n <= high; n++) {
      if (isPerfect(n)) {
         console.log(n);
      }
   }
}

/*
 * Function: isPerfect
 * -------------------
 * isPerfect returns true if and only if the provided number, assumed to be a
 * positive whole number, is perfect.  Restated, isPerfect identifies all of
 * n's proper divisors, sums them all together, and returns true iff that sum
 * incidentally equals n.
 */
function isPerfect(n) {
   var sum = 0;
   for (var factor = 1; factor < n; factor++) {
      if (isDivisibleBy(n, factor)) {
         sum += factor;
      }
   }
   return sum === n;
}
```