Mechanics of Functions

Abraham Gebrekidan
EIP 01
August 07, 2024

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 function calls, we will use 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	nickel + dime	dime + quarter	quarter + dollar
penny + dime	nickel + quarter	dime + dollar	
penny + quarter	nickel + dollar		
penny + 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

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

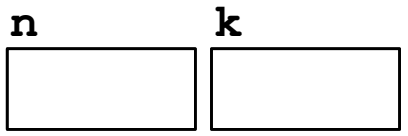
- Given that you already have a **`fact`** function, is easy to turn this formula directly into a function, as follows:

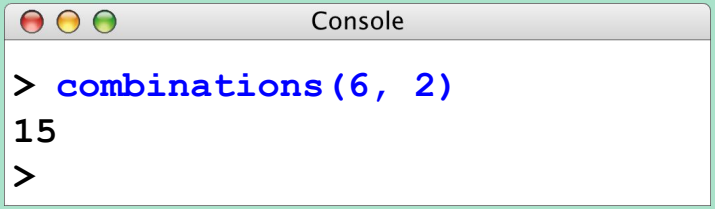
```
function combinations(n, k) {
   return fact(n) / (fact(k) * fact(n - k));
}
```

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

Tracing the `combinations` Function

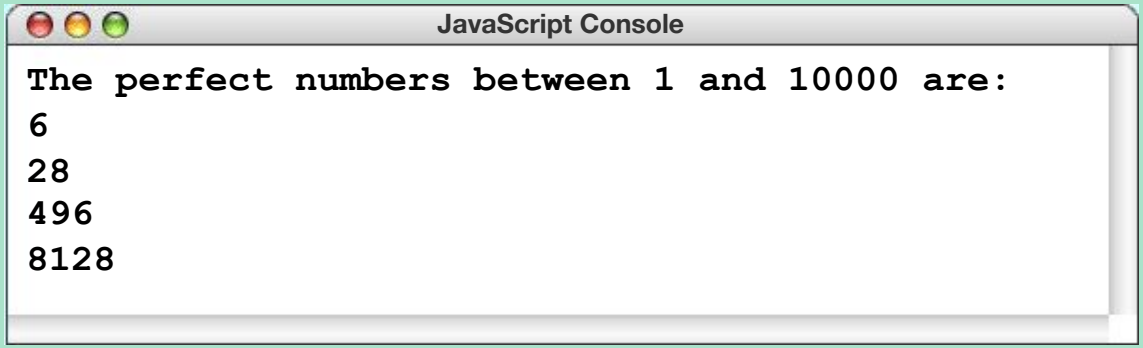
```
function combinations(n, k) {
   return fact(n) / ( fact(k) * fact(n - k) );
}
```





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:



PerfectNumbers.js

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;
}
```

Exercise: Generating Prime Factorizations

- A more computationally intense problem is to generate the prime factorization of a positive integer *n*.
- An integer is prime if it's greater than 1 and has no positive integer divisors other than 1 and itself.
  - ✔ 5 is prime: it's divisible only by 1 and 5.
  - ✔ 6 is not prime: it's divisible by 1, 2, 3, and itself.
- Some prime factorizations:

Console

```
-> PrimeFactorizations(501, 512)
501 = 3 * 167
502 = 2 * 251
503 = 503
504 = 2 * 2 * 2 * 3 * 3 * 7
505 = 5 * 101
506 = 2 * 11 * 23
507 = 3 * 13 * 13
508 = 2 * 2 * 127
509 = 509
510 = 2 * 3 * 5 * 17
511 = 7 * 73
512 = 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2
->
```

PrimeFactorizations.js

PrimeFactorizations.js

```
/*
 * File: PrimeFactorizations
 * -------------------------
 * Produces a table of the prime factorizations for all of the
 * numbers between low and high, inclusive.  For instance,
 * PrimeFactorizations(25, 30) would publish the following
 * to the SJS console.
 *
 *   25 = 5 * 5
 *   26 = 2 * 13
 *   27 = 3 * 3 * 3
 *   28 = 2 * 2 * 7
 *   29 = 29
 *   30 = 2 * 3 * 5
 */

function PrimeFactorizations(low, high) {
   for (var n = low; n <= high; n++) {
      console.log(constructFactorization(n));
   }
}
```

- **Remember to decompose the problem, not the program.**
- **Synthesizing a sequence of prime factorizations is much easier if you operate as if you have a function that synthesizes one.**
- **Invent a series of milestones that advance you towards your overall goal. Each milestone should be a small perturbation to the last fully functional milestone you successfully implemented.**
- **The program you see to the left does *something* if `constructFactorization` produces *something*.**
- **My first milestone? A `for` loop that prints something on behalf of all numbers between `low` and `high`, inclusive.**
- **Invent a placeholder implementation of `constructFactorization` that returns a gesture to what's ultimately needed, and call it progress towards your overall goal.**

PrimeFactorizations.js

PrimeFactorizations.js

```
/*
 * Function: constructFactorization
 * --------------------------------
 * Computes the prime factorization of the supplied
 * number and returns that factorization as a string.
 * The incoming parameter called n is assumed to be
 * positive.
 */
function constructFactorization(n) {
   var result = n + " = ";
   var first = true;
   var factor = 2;

   while (n > 1) {
      if (isDivisibleBy(n, factor)) {
         if (!first) result += " * ";
         first = false;
         result += factor;
         n /= factor;
      } else {
         factor++;
      }
   }

   return result;
}
```

n	180
result	"180 = "
first	true
factor	2

Console

```
-> constructFactorization(180)
```

PrimeFactorizations.js

PrimeFactorizations.js

```
/*
 * Function: constructFactorization
 * --------------------------------
 * Computes the prime factorization of the supplied
 * number and returns that factorization as a string.
 * The incoming parameter called n is assumed to be
 * positive.
 */
function constructFactorization(n) {
   var result = n + " = ";
   var first = true;
   var factor = 2;

   while (n > 1) {
      if (isDivisibleBy(n, factor)) {
         if (!first) result += " * ";
         first = false;
         result += factor;
         n /= factor;
      } else {
         factor++;
      }
   }

   return result;
}
```

n	1
result	"180 = 2 * 2 * 3 * 3 * 5"
first	false
factor	5

Console

```
-> constructFactorization(180)
180 = 2 * 2 * 3 * 3 * 5
->
```

PrimeFactorizations.js

PrimeFactorizations.js

```
/*
 * Function: constructFactorization
 * --------------------------------
 * Computes the prime factorization of the supplied
 * number and returns that factorization as a string.
 * The incoming parameter called n is assumed to be
 * positive.
 */
function constructFactorization(n) {
   var result = n + " = ";
   var first = true;
   var factor = 2;

   while (n > 1) {
      if (isDivisibleBy(n, factor)) {
         if (!first) result += " * ";
         first = false;
         result += factor;
         n /= factor;
      } else {
         factor++;
      }
   }

   return result;
}
```

n	1
result	"180 = 2 * 2 * 3 * 3 * 5"
first	false
factor	5

Console

```
-> constructFactorization(180)
180 = 2 * 2 * 3 * 3 * 5
->
```

PrimeFactorizations.js

PrimeFactorizations.js

```
/*
 * Function: constructFactorization
 * --------------------------------
 * Computes the prime factorization of the supplied
 * number and returns that factorization as a string.
 * The incoming parameter called n is assumed to be
 * positive.
 */
function constructFactorization(n) {
   var result = n + " = ";
   var first = true;
   var factor = 2;

   while (n > 1) {
      if (isDivisibleBy(n, factor)) {
         if (!first) result += " * ";
         first = false;
         result += factor;
         n /= factor;
      } else {
         factor++;
      }
   }

   return result;
}
```

n	1
result	"180 = 2 * 2 * 3 * 3 * 5"
first	false
factor	5

Console

```
-> constructFactorization(180)
180 = 2 * 2 * 3 * 3 * 5
->
```

PrimeFactorizations.js

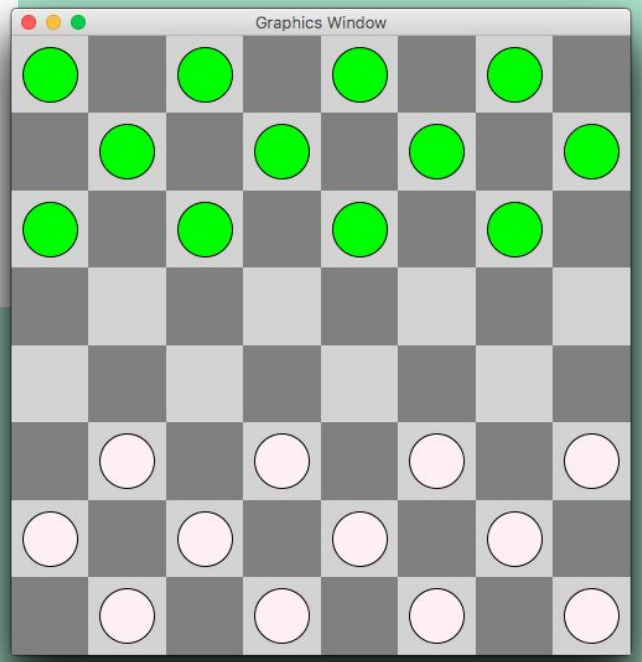
Some thought questions and exercises:

- The solution relies on a single Boolean called **`first`**. What problem is **`first`** solving for us?
- During our trace of **`constructFactorization(180)`**, **`factor`** assumed the values of 2, 3, 4, and 5. 2, 3, and 5 are prime numbers and therefore qualified to appear in a factorization? How does the implementation guarantee 4 will never make an appearance in the returned factorization?
- What is returned by **`constructFactorization(1)`**? How could you have changed the implementation to return **`"1 = 1"`** as a special case return value?
- Trace through the execution of **`constructFactorization(363)`** as we did for **`constructFactorization(180)`**.
- Our implementation relies on a parameter named **`n`** to accept a value from the caller, and then proceeds to destroy **`n`** by repeatedly dividing it down to 1. Does this destruction of **`n`** confuse **`PrimeFactorizations`**'s **`for`** loop? Note that its counting variable is also named **`n`**.

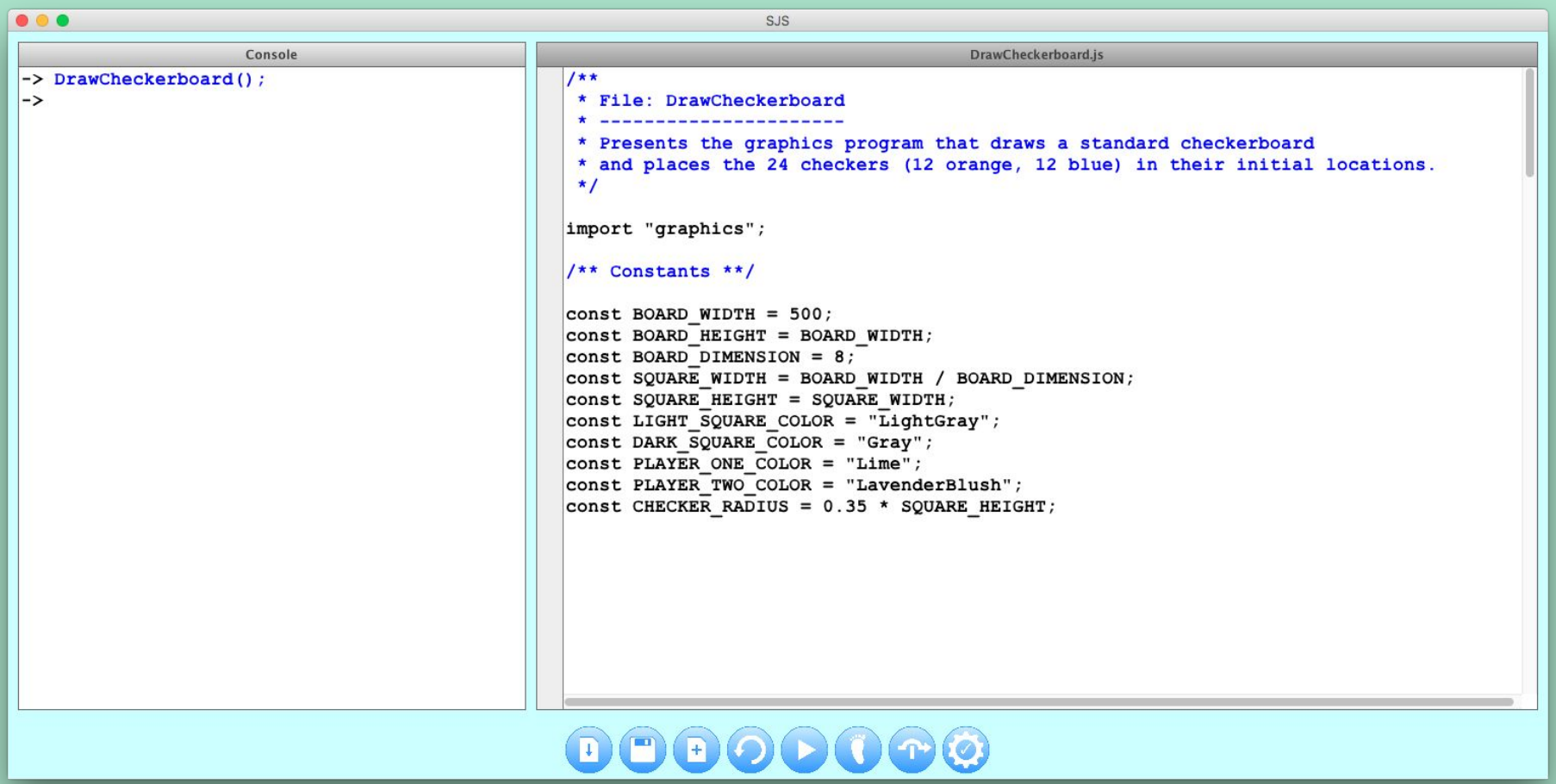
Exercise: Drawing A Checkerboard

- For the rest of lecture, we'll collectively design and decompose (and to the extent we have time, implement) a graphics program that draws the initial configuration for a game of checkers.

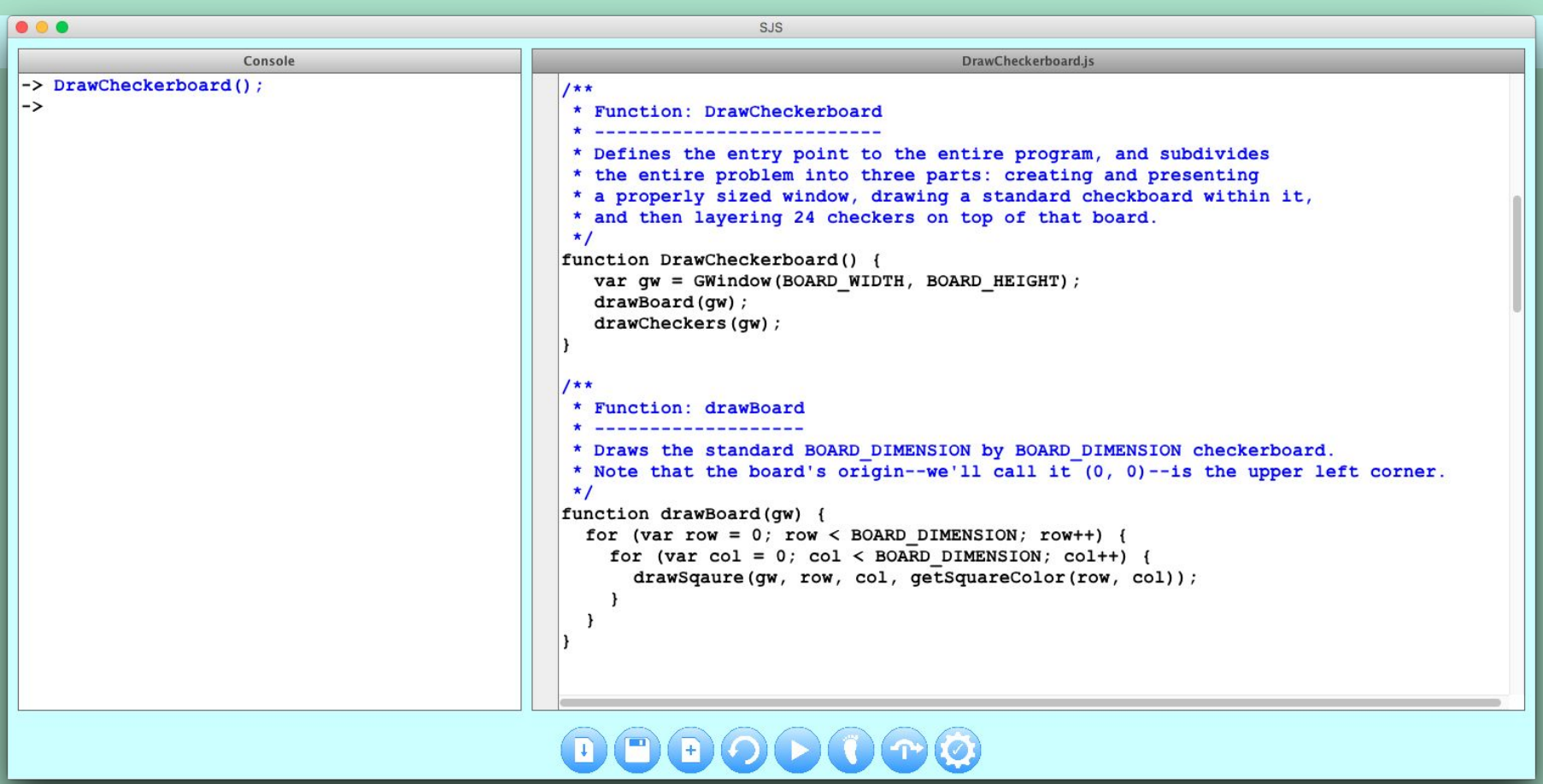
```
/**
 * Function: DrawCheckerboard
 * --------------------------
 * Defines the entry point to the entire program, and subdivides
 * the entire problem into three parts: creating and presenting
 * a properly sized window, drawing a standard checkboard within it,
 * and then layering 24 checkers on top of that board.
 */
function DrawCheckerboard() {
   var gw = GWindow(BOARD_WIDTH, BOARD_HEIGHT);
   drawBoard(gw);
   drawCheckers(gw);
}
```



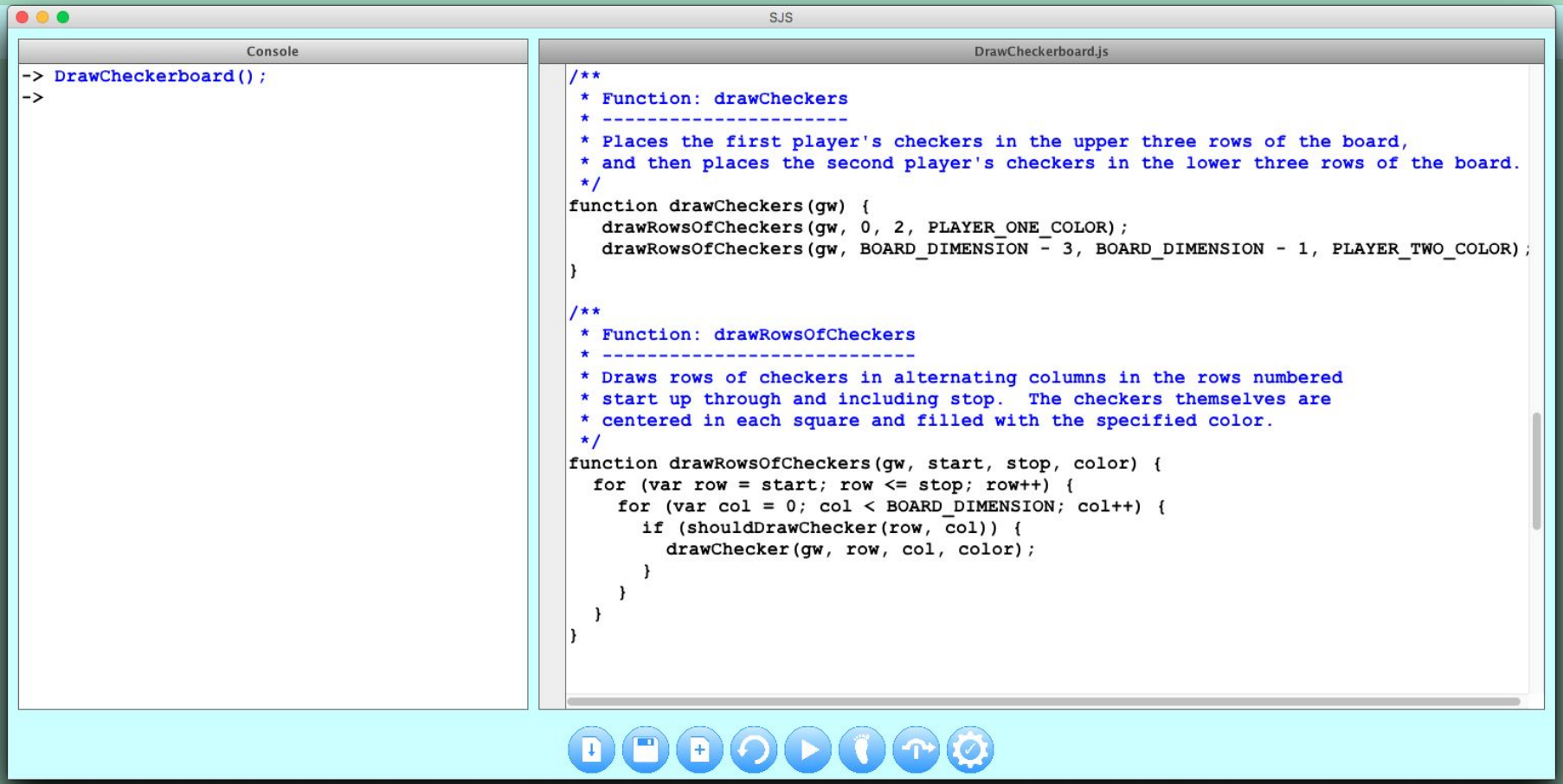
DrawCheckerboard.js



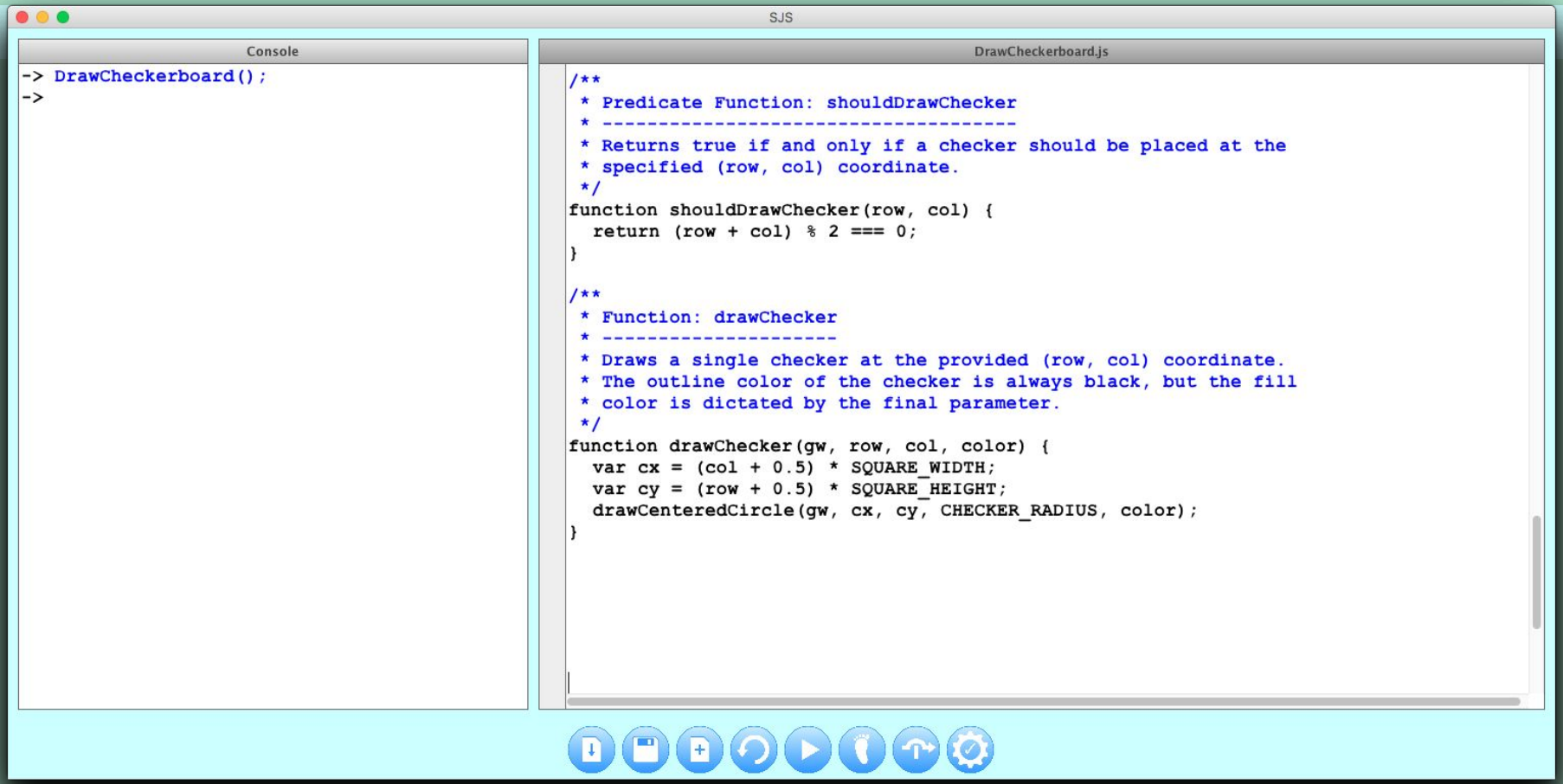
DrawCheckerboard.js



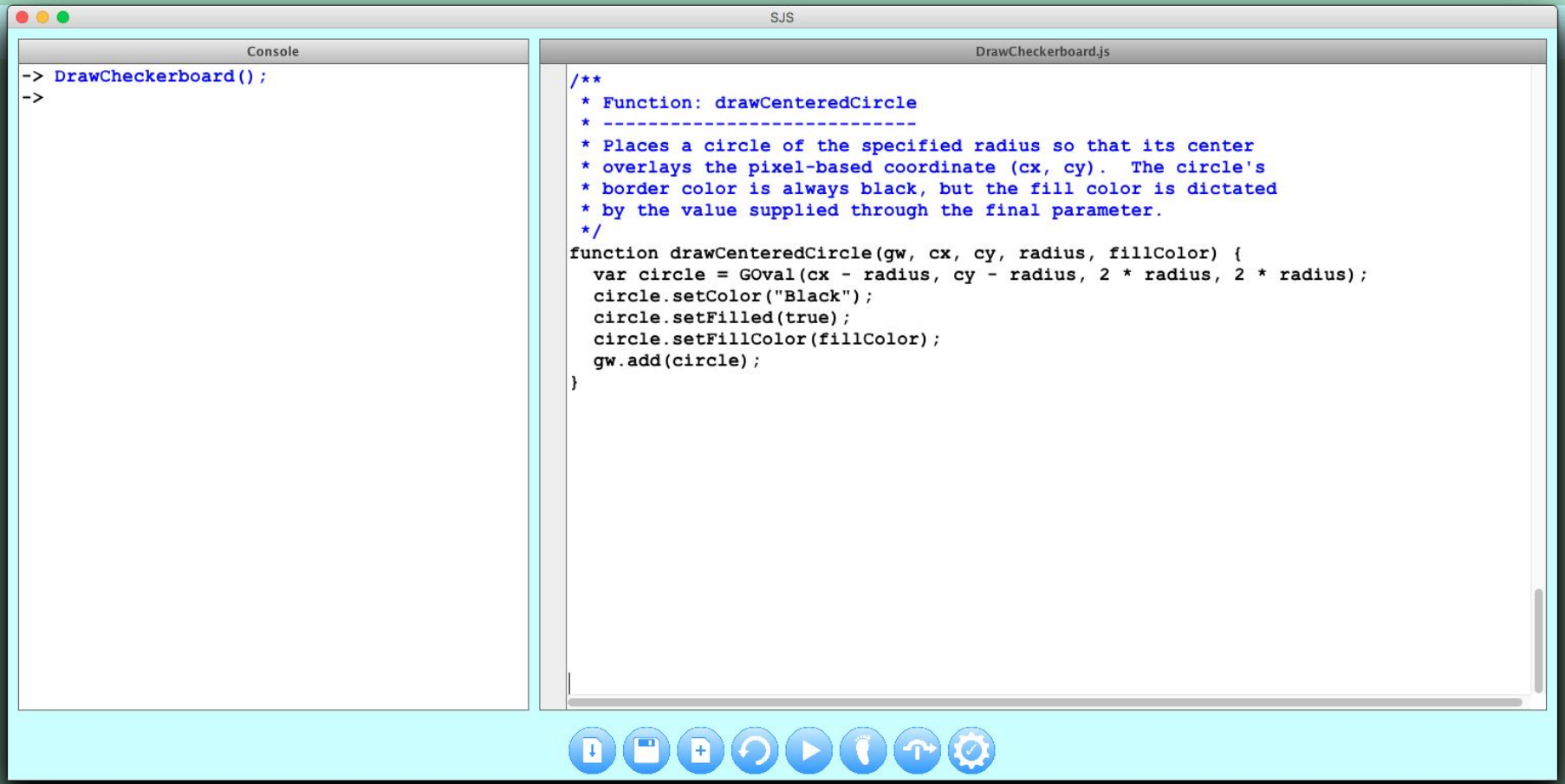
DrawCheckerboard.js



DrawCheckerboard.js



DrawCheckerboard.js



The End