Abraham Gebrekidan
EIP 01

Reading #13A
August 09, 2024

# Mechanics of Functions (Addendum)

## Mechanics of Functions (Addendum)

Abraham Gebrekidan
EIP 01
August 07, 2024

## 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

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`**.

## 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;
}
```

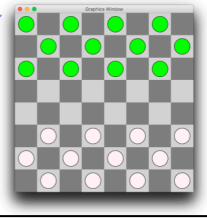
n: 1
result: "180 = 2 * 2 * 3 * 3 * 5"
first: false
factor: 5

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

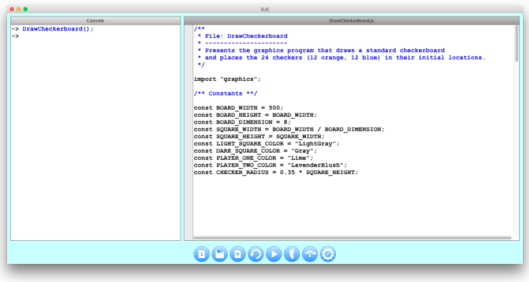
## Exercise: Drawing A Checkerboard

- 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.

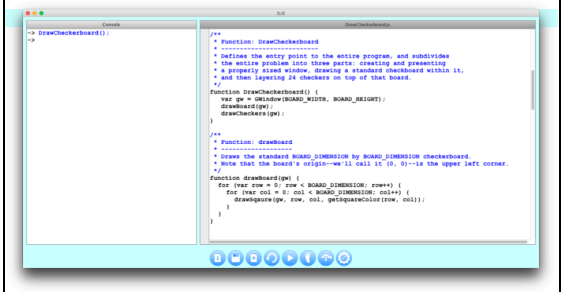
```
/**
 * Function: DrawCheckerboard
 * --------------------------
 * Defines the entry point to the entire program, and subdivides
 * the entire problem into three parts: creating and presenting
 * a properly sized window, drawing a standard checkboard within it,
 * and then layering 24 checkers on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




## DrawCheckerboard.js



# DrawCheckerboard.js

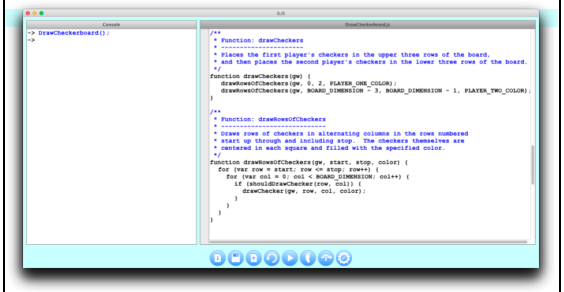


```
-> DrawCheckerboard();
->
```

```
/**
 * 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);
}

/**
 * Function: drawBoard
 * -------------------
 * Draws the standard BOARD_DIMENSION by BOARD_DIMENSION checkerboard.
 * Note that the board's origin--we'll call it (0, 0)--is the upper left corner.
 */
function drawBoard(gw) {
  for (var row = 0; row < BOARD_DIMENSION; row++) {
    for (var col = 0; col < BOARD_DIMENSION; col++) {
      drawSquare(gw, row, col, getSquareColor(row, col));
    }
  }
}
```

# DrawCheckerboard.js

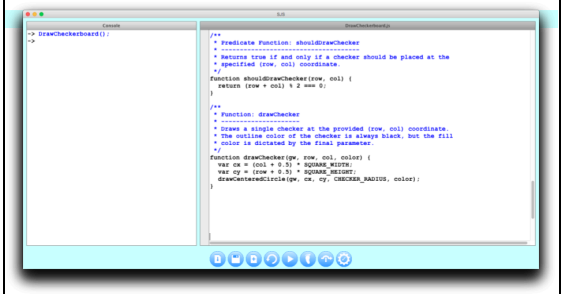


```
-> DrawCheckerboard();
->
```

```
/**
 * Function: drawCheckers
 * ----------------------
 * Places the first player's checkers in the upper three rows of the board,
 * and then places the second player's checkers in the lower three rows of the board.
 */
function drawCheckers(gw) {
  drawRowsOfCheckers(gw, 0, 2, PLAYER_ONE_COLOR);
  drawRowsOfCheckers(gw, BOARD_DIMENSION - 3, BOARD_DIMENSION - 1, PLAYER_TWO_COLOR);
}

/**
 * Function: drawRowsOfCheckers
 * ----------------------------
 * Draws rows of checkers in alternating columns in the rows numbered
 * start up through and including stop.  The checkers themselves are
 * centered in each square and filled with the specified color.
 */
function drawRowsOfCheckers(gw, start, stop, color) {
  for (var row = start; row <= stop; row++) {
    for (var col = 0; col < BOARD_DIMENSION; col++) {
      if (shouldDrawChecker(row, col)) {
        drawChecker(gw, row, col, color);
      }
    }
  }
}
```

# DrawCheckerboard.js

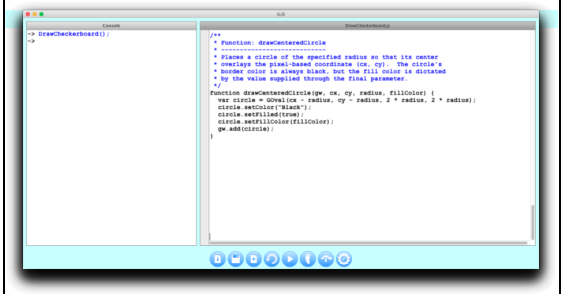


```
-> DrawCheckerboard();
->
```

```
/**
 * Predicate Function: shouldDrawChecker
 * -------------------------------------
 * Returns true if and only if a checker should be placed at the
 * specified (row, col) coordinate.
 */
function shouldDrawChecker(row, col) {
  return (row + col) % 2 === 0;
}

/**
 * Function: drawChecker
 * ---------------------
 * Draws a single checker at the provided (row, col) coordinate.
 * The outline color of the checker is always black, but the fill
 * color is dictated by the final parameter.
 */
function drawChecker(gw, row, col, color) {
  var cx = (col + 0.5) * SQUARE_WIDTH;
  var cy = (row + 0.5) * SQUARE_HEIGHT;
  drawCenteredCircle(gw, cx, cy, CHECKER_RADIUS, color);
}
```

# DrawCheckerboard.js



```
-> DrawCheckerboard();
->
```

```
/**
 * Function: drawCenteredCircle
 * ----------------------------
 * Places a circle of the specified radius so that its center
 * overlays the pixel-based coordinate (cx, cy).  The circle's
 * border color is always black, but the fill color is dictated
 * by the value supplied through the final parameter.
 */
function drawCenteredCircle(gw, cx, cy, radius, fillColor) {
  var circle = GOval(cx - radius, cy - radius, 2 * radius, 2 * radius);
  circle.setColor("Black");
  circle.setFilled(true);
  circle.setFillColor(fillColor);
  gw.add(circle);
}
```

The End