Eric Roberts and Jerry Cain
Handout #15B
CS 106J
April 19, 2017

Mechanics of Functions: Lecture Code

Perfect Numbers

```
/*
 * 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;
}

/*
 * Function: isDivisibleBy
 * -----------------------
 * Returns true if and only if n is divisible by k.
 */
function isDivisibleBy(n, k) {
   return n % k === 0;
}
```

Prime Factorizations

```
/*
 * Program: PrimeFactorizations
 * ----------------------------
 * Produces a table of the prime factorizations for all of the
 * numbers between low and high, inclusive.
 *
 */
function PrimeFactorizations(low, high) {
   for (var n = low; n <= high; n++) {
      console.log(constructFactorization(n));
   }
}

/*
 * 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;
}

/*
 * Function: isDivisibleBy
 * -----------------------
 * Returns true if and only if the second function argument,
 * assumed to be a positive integer, divides evenly into
 * the first (itself assumed to be greater than or equal to 0).
 */
function isDivisibleBy(n, k) {
   return n % k === 0;
}
```

```
/**
 * File: DrawCheckerboard
 * ----------------------
 * Presents the graphics program that draws a standard checkerboard
 * and places the 24 checkers (12 orange, 12 blue) in their initial locations.
 */

import "graphics";

/** Constants **/

const BOARD_WIDTH = 500;
const BOARD_HEIGHT = BOARD_WIDTH;
const BOARD_DIMENSION = 8;
const SQUARE_WIDTH = BOARD_WIDTH / BOARD_DIMENSION;
const SQUARE_HEIGHT = SQUARE_WIDTH;
const LIGHT_SQUARE_COLOR = "LightGray";
const DARK_SQUARE_COLOR = "Gray";
const PLAYER_ONE_COLOR = "Lime";
const PLAYER_TWO_COLOR = "LavenderBlush";
const CHECKER_RADIUS = 0.35 * SQUARE_HEIGHT;

/**
 * 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++) {
         drawSqaure(gw, row, col, getSquareColor(row, col));
      }
   }
}
```

```
/**
 * Function: drawSquare
 * --------------------
 * Draws a single checkerboard square at the specified board coordinate.
 */
function drawSqaure(gw, row, col, color) {
   var ulx = col * SQUARE_WIDTH;
   var uly = row * SQUARE_HEIGHT;
   var square = GRect(ulx, uly, SQUARE_WIDTH, SQUARE_HEIGHT);
   square.setColor(color);
   square.setFilled(true);
   gw.add(square);
}

/**
 * Function: getSquareColor
 * ------------------------
 * Returns one of the two colors used to draw checkerboard squares.
 * Because we want the standard checkerboard pattern, we exploit
 * the modulo-2 characteristics of row + col to decide which of two
 * color constants to return.
 */
function getSquareColor(row, col) {
   return (row + col) % 2 === 0 ? LIGHT_SQUARE_COLOR : DARK_SQUARE_COLOR;
}

/**
 * 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);
         }
      }
   }
}
```

```
/**
 * 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);
}

/**
 * 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);
}
```