



BEN BIN YAN

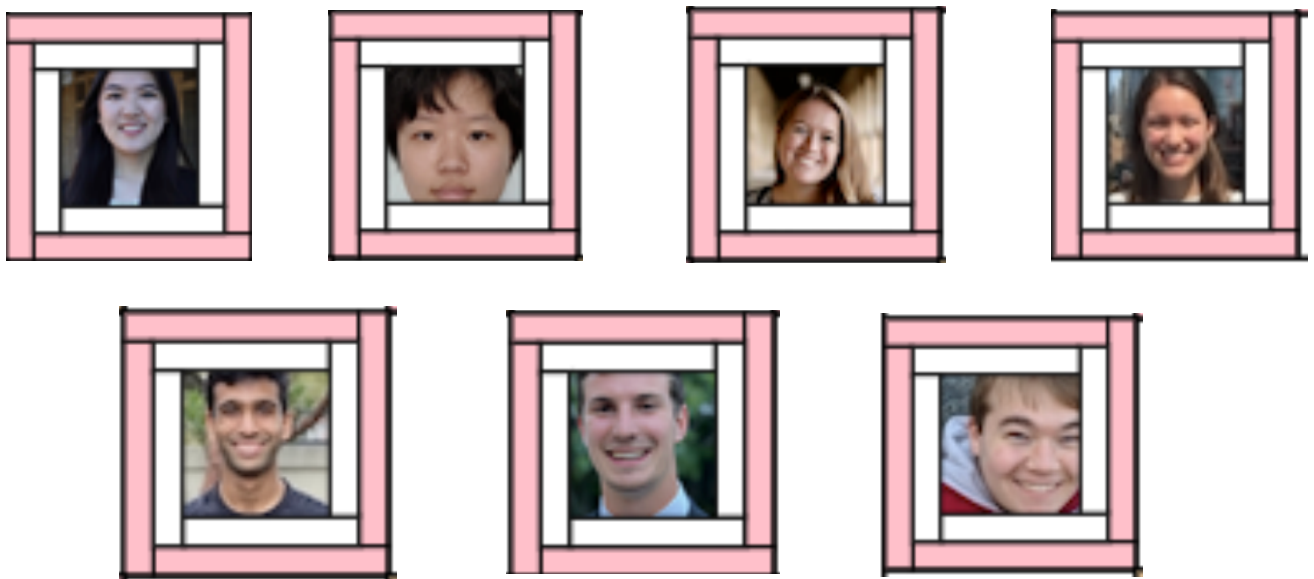
**WHAT
WE TALK
ABOUT
WHEN WE
TALK
ABOUT
JAVASCRIPT**

CS106AX STORIES



“and it ought to make us feel ashamed when we talk like we know what we're talking about when we talk about love.”

— Raymond Carver



Preface and Table of Contents

In anticipation of this year's offering of CS106AX, and because I was extremely bored in the two weeks of summer leading up to this autumn, I decided to try writing a new textbook lol. The resulting creative production, like the *Hobbit* and *Star Wars Prequels* movies, has been obnoxiously split into three parts, a trilogy that no one but your most fervent movie-goers truly asked for. Thankfully, CS106AX students are about as fervent and dedicated as you get in life, so I like to think I understand my audience well ☺.

This is Part I of my course reader, which is concerned with all the elegant, expressive, and beautiful JavaScript—variables, strings, arrays, objects, graphics, animation, interactive mouse-driven programs, and cryptography—that decorates and lights up the first four weeks of the course. Part II, the very creatively titled ***What We Talk About When We Talk About Python***, will be released during the Python segment of our course, and Part III will be released when we begin our final arc of web programming.

Super exciting times ahead! Looking forward to being a part of your learning adventures this autumn ❤️.

Here's a table of contents for ***What We Talk About When We Talk About JavaScript***, my latest short story collection camouflaged as a computer science textbook, and I dearly hope you enjoy! ☺

Short Stories	Start Page
Lecture 1: Intro to HTML and JavaScript	7
Lecture 2: JavaScript Fundamentals	9
Lecture 3: JavaScript Graphics	13
Lecture 4: Interactive Graphics	16
Lecture 5: Timer-Based Animation	19
Lecture 6: Advanced Animation	25
Lecture 7: Strings and GLabel	31
Lecture 8: Arrays	37
Lecture 9: JSON and Objects	43
Lecture 10: ASCII and Unicode	51
Lecture 11: Cryptography	54
JavaScript Syntax Reference 📖	59

Note that in this version of the course reader (for midterm review), **I've omitted the graphics library and random library reference documentation, and instead uploaded them as separate PDFs to the website:**

<https://web.stanford.edu/class/cs106ax/exams/midterm.html>

Note: *No artificial intelligence was used in the writing of this course reader. The only person I ripped off is Jerry Cain ☺. All characters appearing in this work are fictitious. Any resemblance to real people is purely coincidental.*

Lecture 1: Introduction to CS106AX

Written by Ben Yan, from materials by Jerry Cain

Welcome! 🙌

Hi, I'm Ben, your TA for the quarter, and so excited that you've enrolled in, or thinking about enrolling in CS106AX! I hope your first week is off to an electric start, and I look forward to getting to know you!

Please feel free to reach out to me at any point! ☺ I have office hours, am available over email at either bbyan@stanford.edu or judobenlingyan@stanford.edu, and always happy to chat after class.

↳ *Got a little too invested in the World Cup oops...*

HTML, CSS, JavaScript

Modern web applications rely on a triangle of programming technologies: HTML, CSS, and JavaScript.

- **HTML** is the bones and skeleton of a webpage, defining its basic content and structure hierarchy.
- **CSS** is used to regulate color, appearance, and formatting, i.e. making the HTML fashionable.
- **JavaScript (JS)** makes the components above truly interactive, enabling it to operate like a fully-fledged desktop application that just happens to run in a browser. Over 90% of websites use **JS**!

The server that delivers the webpage content is often implemented in **Python**; hence, we learn it later!

HTML File

HTML is short for **Hypertext Markup Language**, and you've likely heard of it if you're chronically online like me, or spent any time in school computer lab vandalizing websites with the *Inspect Element* tool ☺.

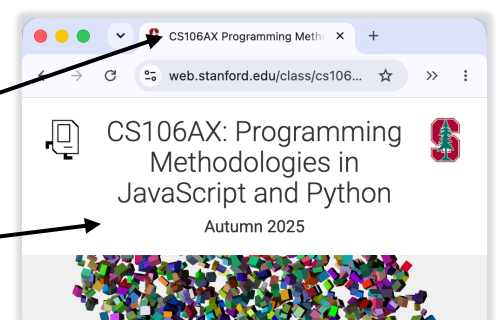
An **HTML file** consists of the **raw text to be displayed on the webpage**, interspersed with various formatting directives that are enclosed in angle brackets, called **HTML tags**, e.g., `<body>`, `<title>`.

HTML tags generally occur in pairs #twinning. The opening tag begins with the name of the tag. The corresponding closing tag has the same name, but preceded with a slash, e.g., `</body>`, `</title>`. The effect of the tag applies to everything between the opening and closing tags. For instance,

```
<title> CS106AX Autumn 2026 </title>
```

Here's a template of what a standard `index.html` file looks like. **This is the naming convention for the homepage of a website**. For instance, when you search up `www.cs106ax.stanford.edu`, you're actually transported to the webpage at the URL `www.cs106ax.stanford.edu/index.html`.

```
<!DOCTYPE HTML>
<html>
  <head>
    <title> title of the page </title>
    Script tags to load JavaScript code
  </head>
  <body>
    Contents of the page, if any.
  </body>
</html>
```



This template features the only HTML tags you'll need for now. Here are standard HTML principles:

- Every file begins with a `<!DOCTYPE html>` tag. This is their way of saying, *"Hello world, I'm an HTML file! ☺"*, just like me saying, especially often this week,
- The entire content is enclosed in `<html>` and `</html>` tags.
- The file begins with a `<head>` section that specifies the **title** (which appears in the browser tab, not on the page), **JavaScript files to load**, and other details not directly visible on the page itself.
- The file includes a `<body>` section that specifies what's physically on the page.

Hello world, my name is Ben!



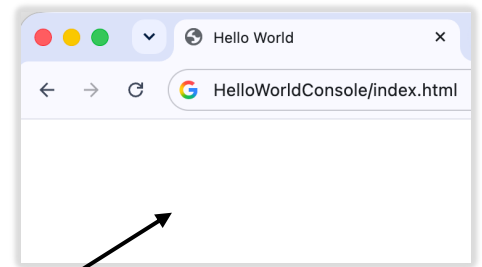
One way of involving JavaScript code in an HTML file is by executing a function (from our script tags) once the HTML content is fully loaded onto the page. To do this, we can revise the `<body>` tag as

```
<body onload="function()"> </body>
```

JavaScript: Hello World!

Here's a compact HTML file that loads JavaScript code from a file called `HelloWorld.js`.

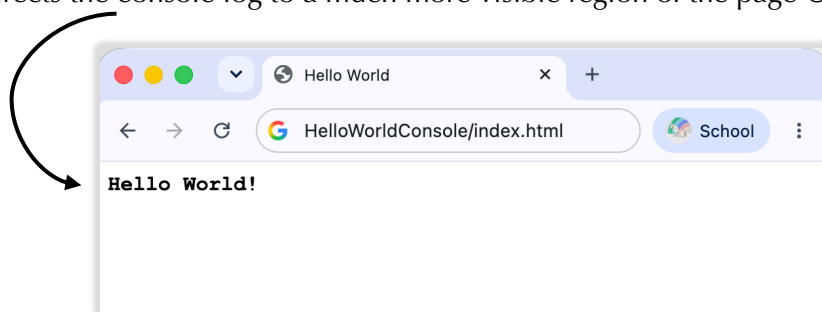
```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello World</title>
    <script text="text/javascript"
      src="HelloWorld.js"></script>
  </head>
  <body onload="HelloWorld()"> </body>
</html>
```



The body of the webpage is initially empty. But once the HTML is loaded, the file asks the browser, through `onload`, to call the function `HelloWorld`. What does this function do? Glad you asked ☺.

```
function HelloWorld() {
  console.log("Hello World!"); // inspired by the first ever C program
}
```

This displays the text `"Hello World!"` on the console log, which appears in different places in different browsers, and can be troublesome to find. For now, we provide a short and sweet library called `JSConsole.js`, which redirects the console log to a much more visible region of the page ☺. That is,



And ta-da, we're done! 🎉

Lecture 2: JavaScript Fundamentals

Written by Ben Yan, from materials by Jerry Cain

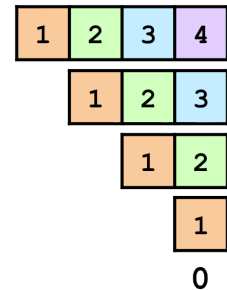
Digit Sum

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.

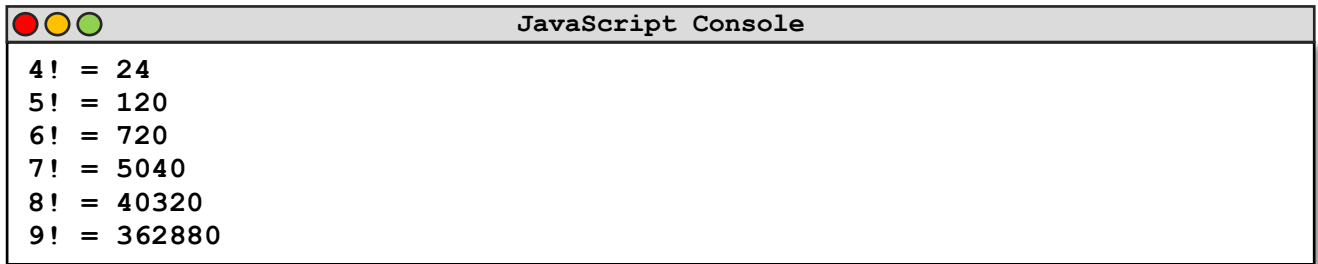
An interesting introductory example is a **for** loop that **yields the factorial of a number n** , computed by multiplying together all positive integers up to and including n , i.e. $n! = 1 * 2 * 3 * \dots * n$.

```
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:



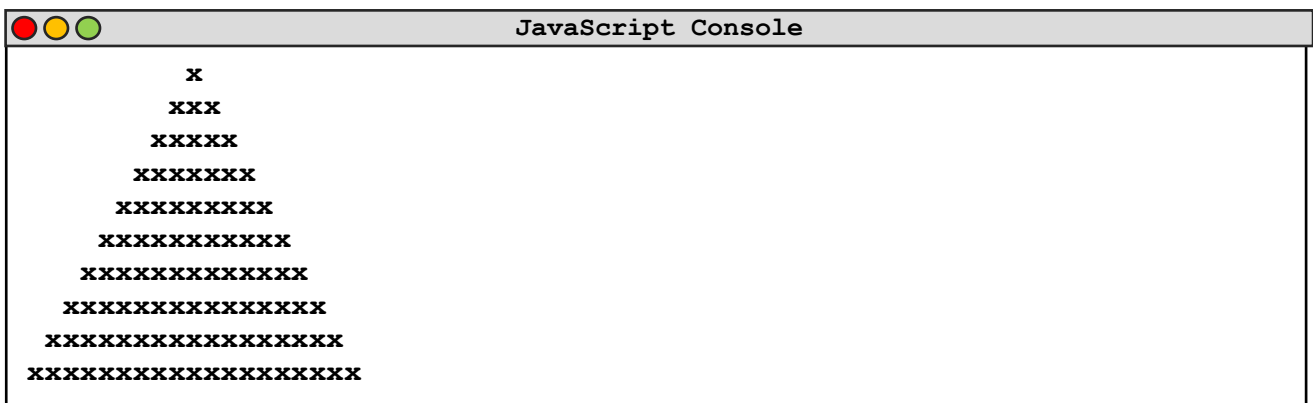
```
JavaScript Console
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
```

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 ☺.

Lecture 3: JavaScript Graphics

Written by Ben Yan, from materials by Jerry Cain

CS106AX JSGraphics Library

This lecture's code—centered around visual and graphical programs—relies on a delightful, in-house library **JSGraphics.js**. This library can be summoned by adding a single line to our HTML files, i.e.

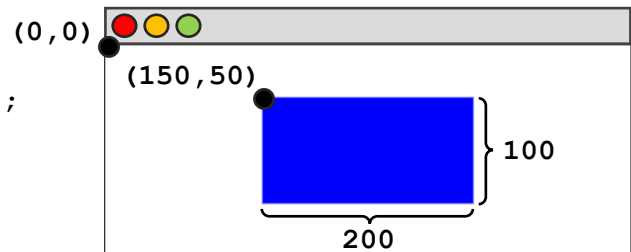
```
<html>
  <head>
    <script type='text/javascript' src='JSGraphics.js'></script>
  </head>
</html>
```

Blue Rectangle

The following program first defines a graphics window or canvas measuring **500** pixels in width and **200** pixels in height. We then create a rectangle of width **200** pixels and height **100** pixels, painted with a solid blue fill, and the rectangle's **top-left corner** placed at coordinate **(x, y) = (150, 50)**.

Note that unlike traditional math, **JavaScript defines the origin (0, 0) as the upper-left corner of the window**, and **values for the y coordinate increase as you go downward** 😊. Weird, but consistent!

```
function BlueRectangle() {
  let gw = GWindow(500, 200);
  let rect = GRect(150, 50, 200, 100);
  rect.setColor("Blue");
  rect.setFilled(true);
  gw.add(rect);
}
```



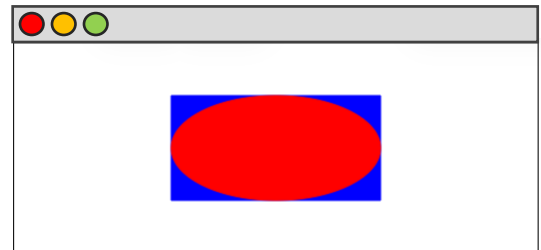
Note the eponymous methods used to create and add the **GWindow** and **GRect** objects:

- `let gw = GWindow(width, height)`
- `let rect = GRect(x, y, width, height)` or `GRect(width, height)`
- `gw.add(rect)` or `gw.add(rect, x, y)`, especially if we didn't specify coordinates above.

Rectangle and Oval

This program extends the one above, by adding a solid red, overlapping **GOval** object to the window, with identical dimensions and top-left coordinates to the rectangle. Note that newer objects can eclipse objects that were added earlier, a layering arrangement called a **stacking order**.

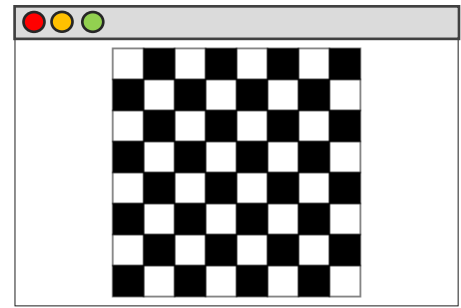
```
function GRectPlusGOval() {
  /* insert here the same code above! */
  let oval = GOval(150, 50, 200, 100);
  oval.setFilled(true);
  oval.setColor("Red");
  gw.add(oval);
} // syntax is similar, GOval(x, y, width, height) or GOval(width, height)
```



Checkerboard

A more visually elaborate and computationally interesting program creates **a grid of filled (black) and non-filled (white) squares to resemble a checkerboard**. The board is centered both horizontally and vertically on the graphics window, just like this.

Assume the following constants have been defined ☺.



```
const GWINDOW_WIDTH = 500;           /* Width of the graphics window */
const GWINDOW_HEIGHT = 300;          /* Height of the graphics window */
const N_COLUMNS = 8;                 /* Number of columns */
const N_ROWS = N_COLUMNS;            /* Number of rows */
const SQUARE_SIZE = 35;              /* Size of a square in pixels */
```

First, we create a graphics window with the specified, not necessarily square dimensions.

```
let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
```

We compute the **top-left coordinate (x0, y0) of the grid**. The full width of the grid is **N_COLUMNS * SQUARE_SIZE**, which leaves a margin of **gw.getWidth() - N_COLUMNS * SQUARE_SIZE**, split equally on the left and right sides. The left margin gives **x0**, and we can calculate **y0** similarly.

```
let x0 = (gw.getWidth() - N_COLUMNS * SQUARE_SIZE) / 2;
let y0 = (gw.getHeight() - N_ROWS * SQUARE_SIZE) / 2;
```

Next, to generate the 2D grid of squares, we'll need to **iterate over each row and column in a doubly nested loop**. For ease, let's focus on a single square at a time—say the one at row **i** and column **j**, for $0 < i < N_ROWS$ and $0 < j < N_COLUMNS$ (note the 0-indexing). Its top-left coordinate is given by

```
let x = x0 + j * SQUARE_SIZE;
let y = y0 + i * SQUARE_SIZE;
```

Note these are offsets from the grid's top-left corner (**x0, y0**). Next, we can create the square **GRect**.

```
let sq = GRect(x, y, SQUARE_SIZE, SQUARE_SIZE);
```

Now, we need to decide if the square should be black (filled) or white (non-filled). **It turns out for all black squares, the row-column sum $i + j$ is even, and for white squares, $i + j$ is odd.** Why is this?

Well, **all squares in one rising diagonal have the same row-column sum, and colors alternate between diagonals**. All squares in the 0th diagonal (just the top-left square) have $i + j = 0$, all squares in the 1st diagonal have $i + j = 1$, all squares in the 2nd diagonal have $i + j = 2$, and so on. As such,

```
let filled = (i + j) % 2 !== 0; // true if black, false if white
sq.setFilled(filled);
gw.add(sq); // don't forget to add the object to the GWindow!
```

And, at long last, we have the main program in its full, checkered glory ☺.

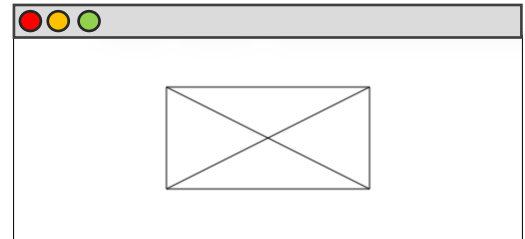
```
function Checkerboard() {
  let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
  let x0 = (gw.getWidth() - N_COLUMNS * SQUARE_SIZE) / 2;
  let y0 = (gw.getHeight() - N_ROWS * SQUARE_SIZE) / 2;

  for (let i = 0; i < N_ROWS; i++) {
    for (let j = 0; j < N_COLUMNS; j++) {
      let x = x0 + j * SQUARE_SIZE;
      let y = y0 + i * SQUARE_SIZE;
      let sq = GRect(x, y, SQUARE_SIZE, SQUARE_SIZE);
      let filled = (i + j) % 2 !== 0;
      sq.setFilled(filled);
      gw.add(sq);
    }
  }
}
```

Crossed Box

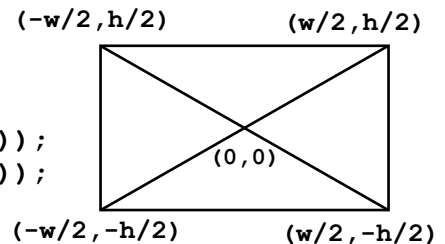
This final program draws an **envelope-looking box with a rectangle and its two diagonal lines**. Our constants are:

```
const GWINDOW_WIDTH = 500; /* in pixels */
const GWINDOW_HEIGHT = 200;
const BOX_WIDTH = 200;
const BOX_HEIGHT = 100;
```



This exercise debuts the **GCompound** class, which makes it possible to combine and layer several **GObjects** into one composite structure. It's akin to a **GWindow** in that you can add objects to it, and akin to a **GObject** in that you can add it to the window. It possesses its own internal coordinate system, and you add objects to the **GCompound** relative to its internal origin. As an example, the function below creates a **GCompound** as architected above, with the diagonals intersecting at the box's center / origin.

```
function createCrossedBox(width, height) {
  let box = GCompound();
  box.add(GRect(-width/2, -height/2, width, height));
  box.add(GLine(-width/2, -height/2, width/2, height/2));
  box.add(GLine(-width/2, height/2, width/2, -height/2));
  return box;
}
```



Note: This makes use of the **GLine(x0, y0, x1, y1)** method to draw a line between two points.

When we add the **GCompound** to the graphics window, we simply set the real coordinates of its local origin—and voilá, all the individual components will appear relative to that point! For example, the function below adds the crossed box so that its internal origin coincides with the center of the canvas.

```
function DrawCrossedBox() {
  let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
  let box = createCrossedBox(BOX_WIDTH, BOX_HEIGHT);
  gw.add(box, gw.getWidth() / 2, gw.getHeight() / 2);
}
```

Lecture 4: Interactive Graphics

Written by Ben Yan, from materials by Jerry Cain

Event Listener Functions 🎧

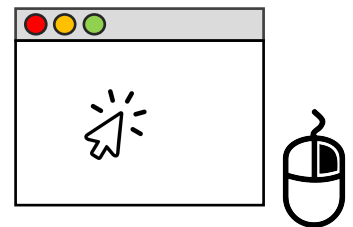
Graphical applications usually make it possible for a user to interface with and even direct a program's actions by using an input device such as a mouse. Programs that support this are considered *interactive*.

As terminology, user actions (e.g., *mouse clicks*, *keystrokes*) are called **events**, and programs that respond to these events are called **event-driven**. For this lecture, we'll write functions that aren't called immediately, but rather sit pretty and *listen* in the background for a particular event type. **Only when that event occurs, the listener or callback function is called.** The general JavaScript template for this is:

```
let eventAction = function(e) {
    code that executes each time this event occurs
}
gw.addEventListener(event, eventAction);
```

For example, if **event** is **"click"**, we'd respond to mouse clicks. More broadly, the **event** types are:

"click"	The user clicks the mouse in the window.
"dblclick"	The user double-clicks the mouse.
"mousedown"	The user presses the mouse button.
"mouseup"	The user releases the mouse button.
"mousemove"	The user glides the mouse with the button up.
"drag"	The user drags the mouse with the button down.



Certain user actions can spark more than one mouse event, e.g., clicking the mouse generates a **"mousedown"** event, a **"mouseup"** event, and a **"click"** event, in that order. Furthermore, each callback function accepts a parameter **e**, which is a data structure with information on the mouse event.

Also note that **function name(parameters)** and **let name = function(parameters)** are equivalent syntax; the latter mainly emphasizes that all functions are assigned to a variable with a name.

Function Closure and Cookie Clicker 🍪

The narrative strength of declaring a function this way—using the assignment operator—occurs when **you declare one function as a local variable inside another function**. That way, the *inner* function can access the *outer* function's local variables, an essential feature for interactive programs as we'll see ☺.

Below is a short program that increments a counter variable, **numCookies**, each time a click occurs.

```
function CookieClicker() { // my own example! not in lecture slides :D
    let numCookies = 0;
    let clickAction = function() {
        numCookies++; // nested function can access this outer variable!
    }
    gw.addEventListener("click", clickAction);
}
```

Draw Dots 🎨

Consider the following interactive program, which draws a dot on the graphics window whenever the user clicks the mouse. Each dot is a filled `GOval` circle centered at the coordinates of the mouse click.

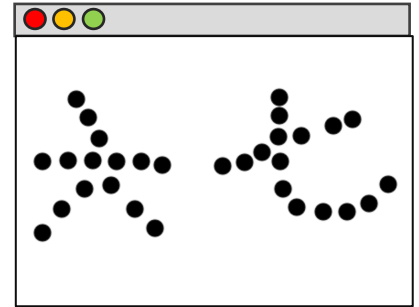
```
/* Constants (in pixels) */
const GWINDOW_WIDTH = 500;
const GWINDOW_HEIGHT = 300;
const DOT_SIZE = 6; // diameter in pixels

function DrawDots() {
    let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);

    let clickAction = function(e) {
        let dotX = e.getX() - DOT_SIZE / 2;
        let dotY = e.getY() - DOT_SIZE / 2;

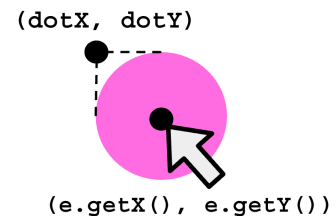
        let dot = GOval(dotX, dotY, DOT_SIZE, DOT_SIZE);
        dot.setFilled(true);
        gw.add(dot);
    };

    gw.addEventListener("click", clickAction);
}
```



The key to understanding this program is the `clickAction` callback function, which is only called into action when the mouse is clicked on-screen. To that end, consider the first 3 lines of its implementation.

```
let clickAction = function(e) {
    let dotX = e.getX() - DOT_SIZE / 2;
    let dotY = e.getY() - DOT_SIZE / 2;
```



The input parameter is mouse event object `e`, which supports two methods you'll want to hammer into memory (**LIKE PLS REMEMBER THEM** 😊). Those are `getX()` and `getY()`, which return the `x` and `y` coordinates of where the click occurred in the `GWindow`'s coordinate system. From there, `e.getX()` and `e.getY()` are offset by the dot's radius, as a `GOval` is positioned by its top-left coordinate.

The rest of its implementation creates the filled circle and adds it to the screen, centered at the click.

```
let dot = GOval(dotX, dotY, DOT_SIZE, DOT_SIZE);
dot.setFilled(true);
gw.add(dot);
```

Crucially, note that `clickAction` has access to the `gw` variable in the outer function `DrawDots`, as `gw` is included in `clickAction`'s *closure* by nesting. A function's **closure** is defined as the combination of its own implementation and all variables available to it, and is a big, big reason this program works 😊.

Draw Lines

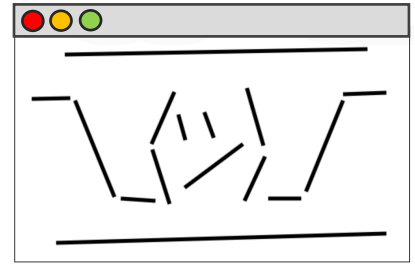
Consider our second interactive graphics program, **DrawLines**, which enables to draw lines with the mouse—by pressing down to create a starting point, and dragging to extend the endpoint as far as we wish.

```
/* Constants */
const GWINDOW_WIDTH = 500;
const GWINDOW_HEIGHT = 300;

function DrawLines() {
  let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
  let line = null;

  let mousedownAction = function(e) {
    line = GLine(e.getX(), e.getY(), e.getX(), e.getY());
    gw.add(line);
  };
  let dragAction = function(e) {
    line.setEndPoint(e.getX(), e.getY());
  };

  gw.addEventListener("mousedown", mousedownAction);
  gw.addEventListener("drag", dragAction);
}
```



First, note the *closure* of the two listener functions, which can both access the window **gw**, and a shared **line** object. These inner functions then work collaboratively to initialize and extend the line on-screen.

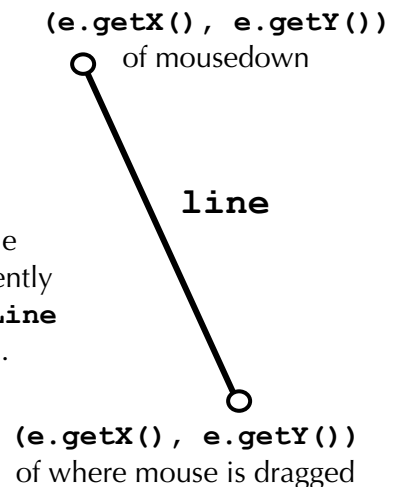
```
let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
let line = null;
let mousedownAction = function(e) { ... }; // called upon "mousedown"
let dragAction = function(e) { ... };      // called upon "drag"
```

The implementation of **mousedownAction** sets the shared variable **line** to a new zero-length line. That is, its start and end points are the same, and both are the coordinates of the mousedown event **e**.

```
let mousedownAction = function(e) {
  line = GLine(e.getX(), e.getY(),
               e.getX(), e.getY());
  gw.add(line);
};
```

The implementation of **dragAction** then takes the **line** object initialized by the “mousedown” handler above, and alters its endpoint to where the mouse is currently located. Note the starting point remains the same as the mousedown, and that **GLine** objects support methods **setStartPoint(x, y)** and **setEndPoint(x, y)**.

```
let dragAction = function(e) {
  line.setEndPoint(e.getX(), e.getY());
};
```



Lecture 5: Timer-Based Animation

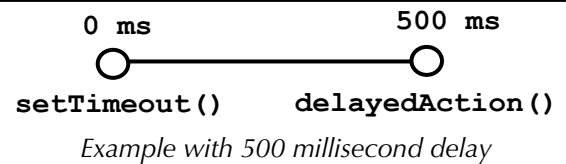
Written by Ben Yan, from materials by Jerry Cain

JavaScript Timers

Last week, we discussed *interactive* programs, which have “callback” functions that respond to mouse events. These events can occur at sudden, irregular times (depending on the user’s actions), or not at all.

Today, we’ll go over *animated* programs, with callback functions that respond to time-scheduled events. There are two types of timers you’ll want to know. The first is a *one-shot timer*, which has this structure

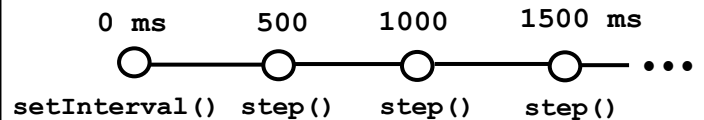
```
let delayedAction = function() {
    block of code that runs after the delay
}
setTimeout(delayedAction, delay);
```



This invokes its callback function, e.g., `delayedAction`, once after the input `delay` in milliseconds.

The second—and more widely used in timer-based animation—is the *interval timer*, with code template

```
let step = function() {
    block of code that runs
    repeatedly at regular intervals
}
let timer = setInterval(step, delay);
```



This timer invokes its callback function, e.g., `step`, not once, but repeatedly and regularly at time intervals of `delay` milliseconds. For instance, if `delay = 500`, then `step` is executed for the 1st time after 500 ms, a 2nd time after 1000 ms, a 3rd time after 1500 ms, and so on. **Note the `setInterval` function returns a `timer` value you can use to terminate it, i.e. by calling `clearInterval(timer)`.**

A metaphor is setting alarms for the morning, in fact many alarms to guarantee you wake up for class ☺. The alarm, of course, doesn’t go off right away when you set it—that would be very *alarming*. Instead, **it’s delayed to ring at 7 AM, then every 10 minutes**, i.e. at 7:10 AM, 7:20 AM, 7:30 AM, so on, until you finally get up and turn off the darn thing. The ring is the `step` function, the timespan between rings is the `delay`, the turn off is `clearInterval`. Here’s this (hopefully not too relatable) metaphor in code.

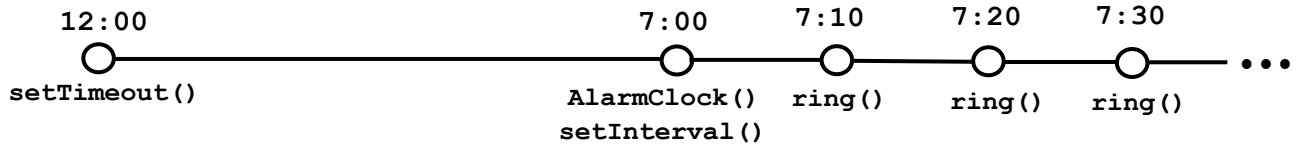
```
const SLEEP_DELAY = 7 * 60 * 60 * 1000; // 7 hours of beauty sleep
const ALARM_DELAY = 10 * 60 * 1000; // time between rings, in milliseconds

function AlarmClock() { // executes only after we sleep 7 hours
    let ring = function() { // executes every 10 min after timeout
        console.log("BEEP BEEP BEEP BEEP!! GET UP BEN!!");
    }
    let timer = setInterval(ring, ALARM_DELAY);
}

setTimeout(AlarmClock, SLEEP_DELAY);
```



The above example uses both the *one-shot* and *interval* timers, each to different effect (one to delay the alarm until morning, the other to ring the alarm repeatedly after the delay), but is absent of any graphics. A caveat is that the alarm first rings at 7:10, not 7:00, due to the initial interval delay, but oh well ☺.

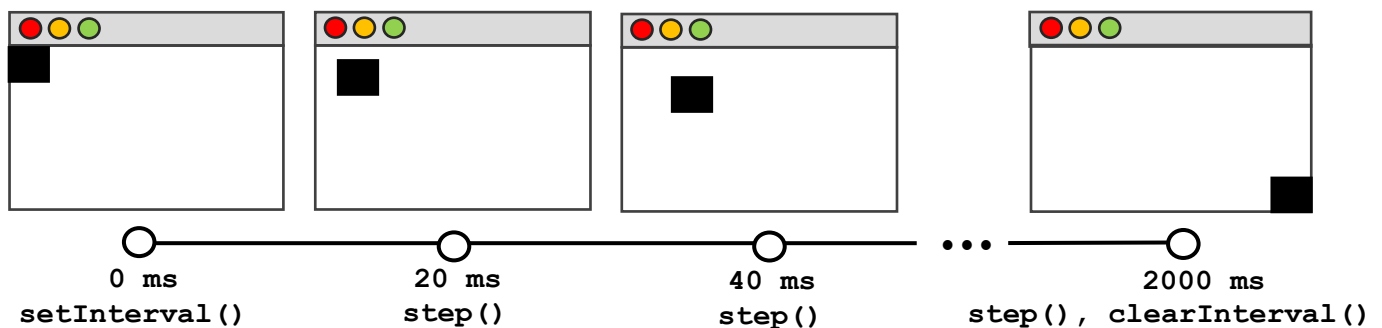


The following programs will focus on **graphics**—and how, by setting a fast-firing **timer**, and having the callback function repeatedly make small updates to **GObjects** on a canvas, we can produce **animation** 🙌. Indeed, in animated films, it's the sequence of evolving frames that creates the illusion of movement.

If the time interval is short enough (typically 20-30 milliseconds), the animation will appear smooth to the naked eye. For cinema or video game enthusiasts, this is equivalent to 33 – 50 frames per second ☺.

Animated Square

Here is an introductory example that animates a square to move diagonally from the top-left corner of the window to the bottom-right corner, as displayed in frames below. It's like the old gliding DVD logo.



Assume the following constants are defined. Note we require the square to take exactly **N_STEPS** = 100 steps from start to end, and that steps occur at regular intervals of **TIME_STEP** = 20 milliseconds.

```
const GWINDOW_WIDTH = 500;
const GWINDOW_HEIGHT = 300;
const N_STEPS = 100;
const TIME_STEP = 20;
const SQUARE_SIZE = 50;
```

First, we'll create the graphics window, and a square with the specified dimensions. The square is placed so that it begins in the top-left corner of the window, as visualized above in the leftmost frame.

```
let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
let square = GRect(0, 0, SQUARE_SIZE, SQUARE_SIZE);
square.setFilled(true);
gw.add(square);
```

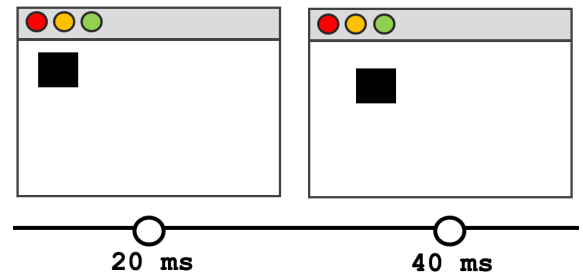
More interestingly, we'll compute **dx** and **dy**, how far the square needs to move per step in the **x** and **y** directions. From start to end, the square's top-left corner goes from (0, 0) to point (w - s, h - s),

for canvas height **h** and width **w**, and square size **s**, all in pixels. To accomplish this in **n** steps, then **dx** = **(w - s) / n**, and **dy** = **(h - s) / n**. In JavaScript, using the above constants, we can write

```
let dx = (gw.getWidth() - SQUARE_SIZE) / N_STEPS;
let dy = (gw.getHeight() - SQUARE_SIZE) / N_STEPS;
```

Here's where the fun begins. We implement the callback **step**, which is called regularly on a schedule of every **TIME_STEP = 20**. Each time step, we move the square by **dx** and **dy**, as calculated above.

```
let stepCount = 0;
let step = function() {
    square.move(dx, dy);
    stepCount++;
    if (stepCount === N_STEPS) {
        clearInterval(timer);
    }
};
let timer = setInterval(step, TIME_STEP);
```



We also keep track of **stepCount** which is declared outside so that all calls to **step** access the same variable in memory. This counter is used to terminate the animation once **N_STEPS** is reached, which means the square has arrived at its destination in the window's bottom-right corner.

Putting this all together, we get the following implementation of **AnimatedSquare** ☺.

```
function AnimatedSquare() {
    let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
    let dx = (gw.getWidth() - SQUARE_SIZE) / N_STEPS;
    let dy = (gw.getHeight() - SQUARE_SIZE) / N_STEPS;

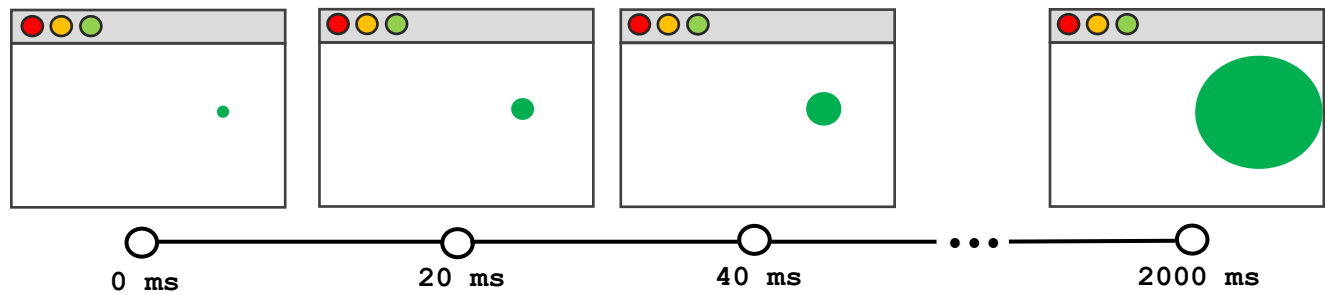
    let square = GRect(0, 0, SQUARE_SIZE, SQUARE_SIZE);
    square.setFilled(true);
    gw.add(square);

    let stepCount = 0;
    let step = function() {
        square.move(dx, dy);
        stepCount++;
        if (stepCount === N_STEPS) {
            clearInterval(timer);
        }
    };
    let timer = setInterval(step, TIME_STEP);
}
```

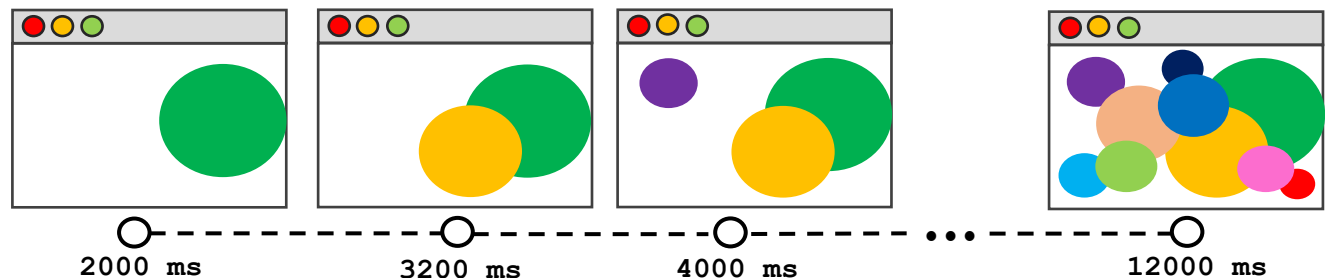
Exploding Circles

We'll write a program **GrowingCircles** that **draws 10 filled, randomly shaded circles at various locations within a graphics window**. The program is ornamented with the following animated features.

For the first circle, its center coordinate and full radius are randomly generated so it fits entirely within the canvas. Initially, we draw the circle with radius zero, but the radius gradually increases over several time steps until the radius matches what we generated previously. An example is displayed below for a circle that grows **1 pixel in diameter** every **20 ms**, until it becomes a fully-fledged **50 pixels in radius**.



Each of the other circles is introduced in the same “exploding” manner as above, but the second circle is only placed after the first circle has fully expanded, the third circle is only placed after the second has fully expanded, and so forth. The animation ends when the tenth and final circle is fully expanded. The timeline below displays a 2nd circle with a 30-pixel radius, a 3rd circle with a 20-pixel radius, and so on.



Assume the following constants are defined. Each circle’s radius is a randomly and independently chosen value between **MIN_RADIUS = 15** and **MAX_RADIUS = 50** pixels. The currently “exploding” circle grows by **DELTA_SIZE = 1** pixels in diameter every **TIME_STEP = 20** milliseconds.

```
const GWINDOW_WIDTH = 500;
const GWINDOW_HEIGHT = 300;
const N_CIRCLES = 10;
const MIN_RADIUS = 15;
const MAX_RADIUS = 50;
const TIME_STEP = 20;
const DELTA_SIZE = 1;
```

First, we create the canvas, and a variable to keep track of how many circles we’ve fully created so far.

```
let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
let circlesCreated = 0;
```

We'll also want to store—and make accessible to callback functions—the current **circle** (a **GOval** object) we're expanding, as well as its current diameter, and the full diameter it should be expanded to.

```
let desiredSize = 0; /* all are initialized to temporary zero values! */
let currentSize = 0;
let circle = null;
```

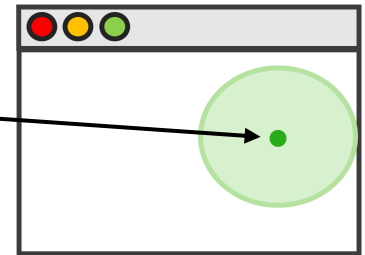
Next, we'll write a helper function that **generates a circle with initially zero radius, but with a random future radius r when fully grown**. The circle's color and center coordinate (x, y) are also random. We assign the **GOval** object to **circle** above, as well as update variables **desiredSize**, **currentSize**.

```
let createNewCircle = function() {
  let r = randomReal(MIN_RADIUS, MAX_RADIUS); // future radius!

  let x = randomReal(r, GWINDOW_WIDTH - r);
  let y = randomReal(r, GWINDOW_HEIGHT - r);

  circle = GOval(x - 0, y - 0, 0, 0);
  circle.setFilled(true);
  circle.setColor(randomColor());

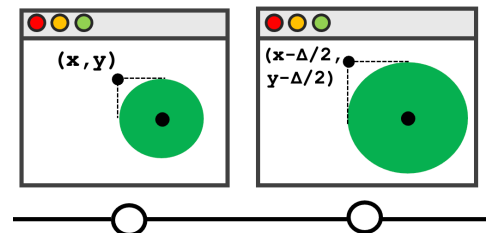
  desiredSize = 2 * r; // note size is diameter
  currentSize = 0;     // not the radius!
  return circle;
}
```



Note for the fully-fledged circle to fit entirely within the window, we must have $r \leq x \leq w - r$ and $r \leq y \leq h - r$, where w and h are the width and height of the graphics window, respectively.

To proceed, we focus on just one circle at a time ☺, and draft a **step** function which grows the current **circle**'s diameter by **DELTA_SIZE** = 1 pixel, as long as the circle hasn't reached its fully grown size.

```
let step = function() { // this is called every TIME_STEP milliseconds!
  if (currentSize < desiredSize) {
    currentSize += DELTA_SIZE;
    let x = circle.getX() - DELTA_SIZE / 2;
    let y = circle.getY() - DELTA_SIZE / 2;
    circle.setBounds(x, y, currentSize, currentSize);
  }
  /* more code to implement later! */
};
let timer = setInterval(step, TIME_STEP);
```



Note that a **GOval**'s location is specified by its top-left coordinate. Thus, **when the circle's diameter increases by Δ pixels**, to maintain the same center, its **top-left coordinate actually moves**—up and to the left by $\Delta / 2$ pixels. We then use **.setBounds($x, y, width, height$)** to alter the circle's location and dimensions to effectively grow the circle outward from its center.

Next, we'll need to handle the **else** condition, i.e. when **currentSize** reaches **desiredSize**. In this case, we should check if there are more circles to grow, or if the animation should be terminated.

```

let step = function() {
  if (currentSize < desiredSize) {
    /* same code as above to grow the circle's diameter! */
  }
  /* we can add a new circle, and there are more circles to grow */
  else if (circlesCreated < N_CIRCLES) {
    gw.add(createNewCircle()); // start growing a new circle
    circlesCreated++;
  } else { // when all circles are fully expanded, stop animation
    clearInterval(timer);
  }
}

```

If we haven't created all circles yet, we enlist `createNewCircle` to generate a new `circle`, and reset `currentSize`, `desiredSize` so the next call to `step()` enters the `if` loop, growing that new circle.

Constellating all the above jigsaw pieces together, with some style ironing, we have our final solution ☺.

```

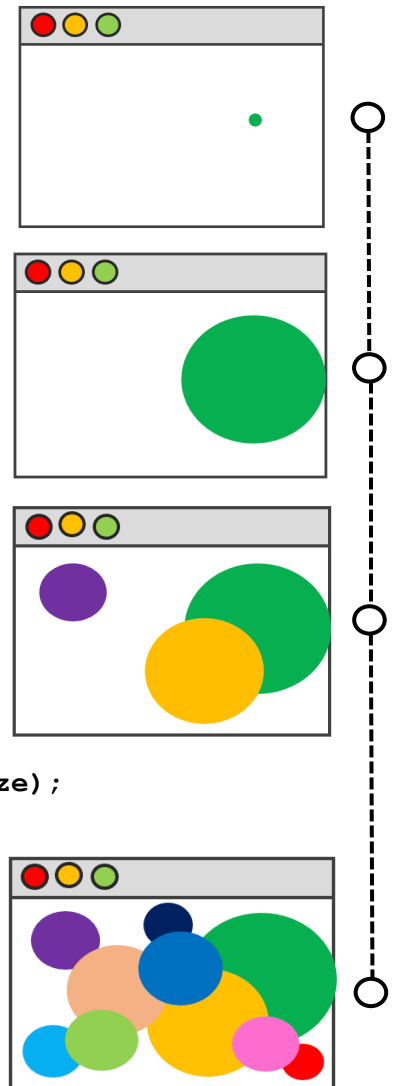
function GrowingCircles() {
  let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
  let circlesCreated = 0;
  let desiredSize = 0;
  let currentSize = 0;
  let circle = null;

  let createNewCircle = function() {
    let r = randomReal(MIN_RADIUS, MAX_RADIUS);
    let x = randomReal(r, GWINDOW_WIDTH - r);
    let y = randomReal(r, GWINDOW_HEIGHT - r);
    circle = GOval(x, y, 0, 0);
    circle.setFilled(true);
    circle.setColor(randomColor());
    desiredSize = 2 * r;
    currentSize = 0;
    return circle;
  };

  let step = function() {
    if (currentSize < desiredSize) {
      currentSize += DELTA_SIZE;
      let x = circle.getX() - DELTA_SIZE / 2;
      let y = circle.getY() - DELTA_SIZE / 2;
      circle.setBounds(x, y, currentSize, currentSize);
    } else if (circlesCreated < N_CIRCLES) {
      gw.add(createNewCircle());
      circlesCreated++;
    } else {
      clearInterval(timer);
    }
  };

  let timer = setInterval(step, TIME_STEP);
}

```



Lecture 6: Advanced Animation

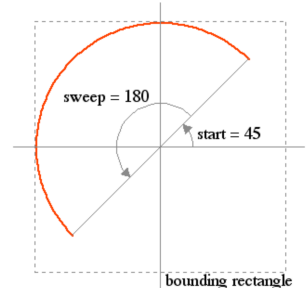
Written by Ben Yan, from materials by Jerry Cain

GArc Geometry

The **GArc** class represents an arc formed by taking a section from the perimeter of an oval. An arc is defined by its **start angle**, its **sweep angle**, and its bounding box, i.e. the top-left coordinate **(x,y)**, and width and height of its corresponding oval.

The following method can be used to create a custom **GArc** object.

```
let arc = GArc(x, y, width, height,
               startAngle, sweepAngle);
```



Angles are measured in degrees going counterclockwise from the positive x axis. Negative angles signify a clockwise direction. Also, if you'd like to procrastinate setting the arc's location, you can write:

```
let arc = GArc(width, height, startAngle, sweepAngle);
gw.add(arc, x, y); // choose location only when we add it to the canvas!
```

The **GArc** class also supports this gallery of methods, which allow you to animate an arc's appearance.

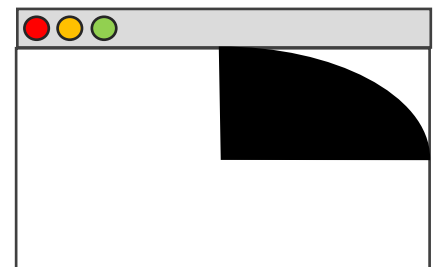
setStartAngle(start)	Sets the start angle for the arc.
getStartAngle()	Returns the start angle for the arc.
setSweepAngle(sweep)	Sets the sweep angle for the arc.
getSweepAngle	Returns the sweep angle of the arc.
setFrameAngle(x, y, width, height)	Resets the bounds for the corresponding oval's frame.

Filled Elliptical Arc

The following program displays a shaded, pie-shaped wedge formed by the center and the endpoints of arc sweeping 90 total degrees.

```
function FilledEllipticalArc() {
    let gw = GWindow(GWINDOW_WIDTH,
                     GWINDOW_HEIGHT);
    let start = 0; // angle in degrees
    let sweep = 90; // angle distance in degrees

    let arc = GArc(0, 0, gw.getWidth(), gw.getHeight(), start, sweep);
    arc.setFilled(true);
    gw.add(arc); // note top-left coordinate of oval is the origin (0,0)!
}
```

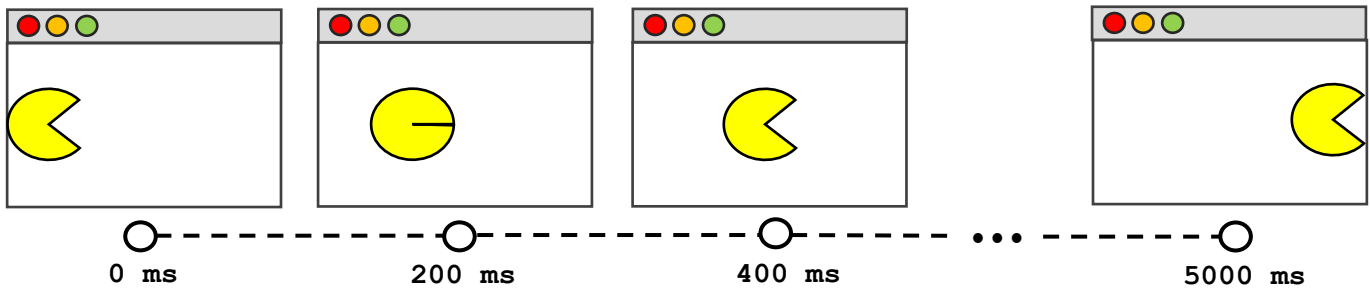


Note the **GArc** class supports functions **setFilled(boolean)** and **setFillColor(color)**, which are similar to their **G Oval** and **GRect** counterparts.

Pac-Man

We'll write a program that uses a **GArc** object to display a **PacMan** figure (like the one from the arcade game), placing him initially at the left edge of the graphics window. From there, a timer animation enables **PacMan** to glide horizontally to the right edge of the window, and as he moves, the program should change the start and sweep angles of the **GArc** so the mouth appears to open and close.

The mouth is fully open when the start and sweep are **45** and **270** degrees, respectively, and the mouth is fully closed when the start and sweep are **0** and **360** degrees (i.e. full circle), as illustrated below.

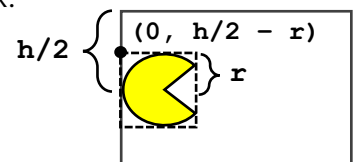


Assume the following constants have been defined. Here, the diameter of **PacMan**'s oval is **60** pixels, and every **20** milliseconds, he moves at **2** pixels in the **x** direction, and the start angle of his arc changes by **DTHETA** = **5** pixels—positively if his mouth is opening, and negatively if his mouth is closing.

```
const GWINDOW_WIDTH = 500;
const GWINDOW_HEIGHT = 300;
const TIME_STEP = 20;
const PACMAN_SIZE = 60; /* diameter in pixels */
const PACMAN_DX = 2;
const PACMAN_DTHETA = 5;
```

First, we define the graphics window, as well as **PacMan**'s initial bounding box.

```
let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
let x0 = 0;
let y0 = gw.getHeight() / 2 - PACMAN_SIZE / 2;
```



Next, we create **PacMan** ☺, placed at his starting location, and with his mouth fully open, ready to eat!!

```
let pacman = GArc(x0, y0, PACMAN_SIZE, PACMAN_SIZE, 45, 270);
pacman.setFilled(true);
pacman.setFillColors("Yellow");
gw.add(pacman);
```

In anticipation of the **step** function that'll animate PacMan to life, we'll want to keep track of two state variables. One is the start **angle** of his mouth. The other is a **sign** value, which is positive (**+1**) if his mouth is opening (**angle** is increasing), and negative (**-1**) if closing (**angle** is decreasing).

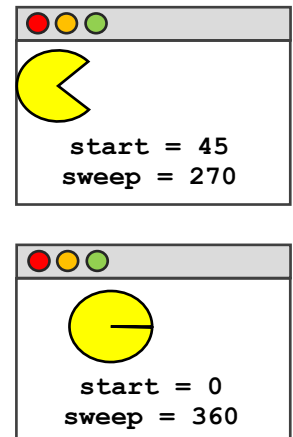
```
let angle = 45;
let sign = -1;
```

We proceed to implement **step**, which is called on regular intervals of **TIME_STEP** milliseconds.

```
let step = function() {
    angle += sign * PACMAN_DTHETA;
    if (angle === 0 || angle === 45) sign = -sign;

    pacman.setStartAngle(angle);
    pacman.setSweepAngle(360 - 2 * angle);
    pacman.move(PACMAN_DX, 0); // horizontal movement

    if (pacman.getX() + PACMAN_SIZE >= gw.getWidth()) {
        clearInterval(timer);
    } // reaches right side of window
};
let timer = setInterval(step, TIME_STEP);
```



In each step, we increase or decrease **angle** by **DTHETA** pixels, and change the start and sweep angles accordingly. Note for his mouth's opening to be vertically symmetric, the end angle must be $360 - \text{angle}$, meaning the sweep angle adjusts to $(360 - \text{angle}) - \text{angle} = 360 - 2 * \text{angle}$.

When the start **angle** reaches either **0** (fully close) or **45** (fully open), we reverse the **sign** so the mouth moves the other way in future steps. Lastly, once the bounding box of **PacMan** kisses the right side of the window, we call **clearInterval** to stop the timer animation.

Assembling all the knobs and screws together, we get this vintage implementation of arcade PacMan ☺.

```
function PacMan() {
    let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
    let x0 = 0;
    let y0 = (gw.getHeight() - PACMAN_SIZE) / 2;

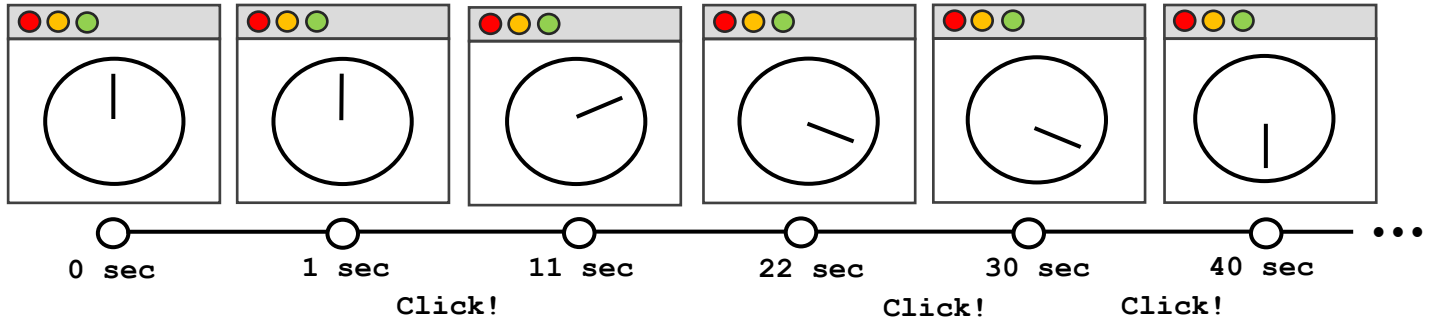
    let pacman = GArc(x0, y0, PACMAN_SIZE, PACMAN_SIZE, 45, 270);
    pacman.setFilled(true);
    pacman.setFill("Yellow");
    gw.add(pacman);

    let angle = 45;
    let sign = -1;
    let step = function() {
        angle += sign * PACMAN_DTHETA;
        if (angle === 0 || angle === 45) sign = -sign;

        pacman.setStartAngle(angle);
        pacman.setSweepAngle(360 - 2 * angle);
        pacman.move(PACMAN_DX, 0);
        if (pacman.getX() + PACMAN_SIZE >= gw.getWidth()) {
            clearInterval(timer);
        }
    };
    let timer = setInterval(step, TIME_STEP);
}
```

Stopwatch

Next, we'll write a graphical program that places a stopwatch so that it's centered within a graphics window. The stopwatch is stationary until the user clicks the mouse anywhere, at which point the stopwatch hand rotates at a speed of 360 degrees per minute. When the user clicks a second time, the stopwatch is halted, but a third click prompts it to continue rotating as if never interrupted, and so on.



Note we'll need a marriage of both mouse and timer events! 🤖 Assume the definition of these constants.

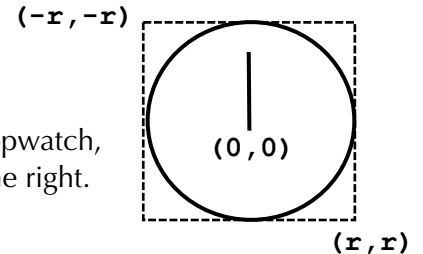
```
const GWINDOW_WIDTH = 500; // in pixels
const GWINDOW_HEIGHT = 300;
const STOPWATCH_RADIUS = 80;
const STEP_TIME = 60000/360; // rotate 360 degrees over course of 60 sec
const DTHETA = 1; // in degrees! change in angle every step
```

We also derive another pair of constants. Note the stopwatch's hand is slightly shorter than its radius.

```
const STOPWATCH_DIAMETER = 2 * STOPWATCH_RADIUS;
const STOPWATCH_HANDLENGTH = 0.8 * STOPWATCH_RADIUS;
```

First, we write a helper function that creates a **GCompound** object for the stopwatch, which consists of a **GOval** and a **GLine** for the watch hand, as shown on the right.

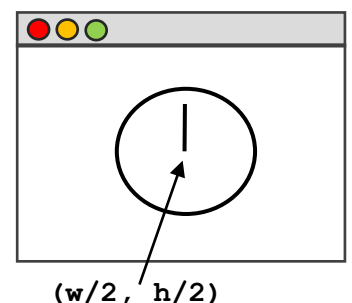
```
function createStopwatch() {
    let stopwatch = GCompound();
    stopwatch.add(GOval(-STOPWATCH_RADIUS, -STOPWATCH_RADIUS,
                        STOPWATCH_DIAMETER, STOPWATCH_DIAMETER));
    stopwatch.add(GLine(0, 0, 0, -STOPWATCH_HANDLENGTH));
    return stopwatch;
}
```



Note: In the coordinate system of **JSGraphics**, the **y** coordinate increases as we go down, not up 😊.

Next, we create the graphics window, and enlist the helper function above to create the stopwatch **GCompound** object, and add it so that its origin or center coincides with the center of the window.

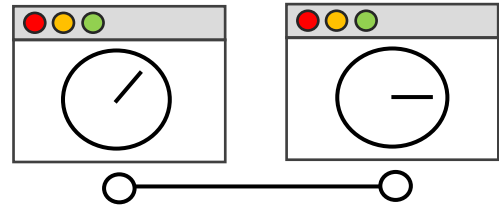
```
let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
let stopwatch = createStopwatch();
gw.add(stopwatch, GWINDOW_WIDTH / 2, GWINDOW_HEIGHT / 2);
```



To proceed, we'll draft an excerpt of code to animate the watch hand to move. We do this by **rotating the entire GCompound about its origin**, using the **rotate(*angle*)** method, where *angle* is in degrees.

```
let step = function() {
    stopwatch.rotate(-DTHETA);
};

let timer = setInterval(step, STEP_TIME);
```



Note that the angle, **-DTHETA**, is negative to rotate **clockwise** instead of counterclockwise, and that **STEP_TIME** is intentionally calculated so the hand rotates a full 360 degrees around in one minute ☺.

Next, we embed this animation as part of a "**click**" event listener—so that a click toggles between stopping the timer (if a timer is currently active), and starting the timer (if the hand is stationary).

```
let timer = 0; // sentinel value that says there's no active timer

let clickAction = function(e) {
    if (timer === 0) {
        // hand is stationary, so start animating it
        let step = function() {
            stopwatch.rotate(-DTHETA);
        };
        timer = setInterval(step, STEP_TIME);

    } else {
        // hand is moving, so make it stationary by stopping animation
        clearInterval(timer);
        timer = 0;
    }
};

gw.addEventListener("click", clickAction);
```

Note that we declare **timer** outside both the event and timer callback functions, so that all calls access, and potentially modify, the same exact **timer** value. Moreover, we use a **0** value as a sentinel or signal that no **timer** is actively running, so that **clickAction** knows whether to start or stop the clock. We alternatively could have defined a Boolean variable, e.g., **clockIsRunning**, and used that instead.

The full solution—with all the moving clock parts and stationery above—is shown on the next page ☺.



Little Ben

"I love clocks!"
– Ben



Big Ben

```

/* File: Stopwatch.js
 * -----
 * This program presents a simple animated stopwatch that can be started,
 * stopped, and continued by repeatedly clicking the mouse.
 */

/* Constants */
const GWINDOW_WIDTH = 500; // in pixels
const GWINDOW_HEIGHT = 300;
const STOPWATCH_RADIUS = 80;
const STEP_TIME = 60000/360; // rotate 360 degrees over course of 60 sec
const DTHETA = 1; // in degrees! change in angle every step

/* Derived Constants */
const STOPWATCH_DIAMETER = 2 * STOPWATCH_RADIUS;
const STOPWATCH_HANDLENGTH = 0.8 * STOPWATCH_RADIUS;

function Stopwatch() {
    let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
    let stopwatch = createStopwatch();
    gw.add(stopwatch, GWINDOW_WIDTH/2, GWINDOW_HEIGHT/2);

    let timer = 0; // sentinel value that says there's no active timer

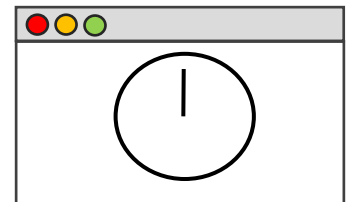
    let clickAction = function(e) {
        if (timer === 0) {
            // hand is stationary, so start animating it
            let step = function() {
                stopwatch.rotate(-DTHETA);
            };
            timer = setInterval(step, STEP_TIME);
        } else {
            // hand is moving, so make it stationary by stopping timer
            clearInterval(timer);
            timer = 0;
        }
    };
    gw.addEventListener("click", clickAction);
}

function createStopwatch() {
    let stopwatch = GCompound();

    stopwatch.add(GOval(-STOPWATCH_RADIUS, -STOPWATCH_RADIUS,
                        STOPWATCH_DIAMETER, STOPWATCH_DIAMETER));
    stopwatch.add(GLine(0, 0, 0, -STOPWATCH_HANDLENGTH));

    return stopwatch;
}

```



Lecture 7: Strings in JavaScript

Written by Ben Yan, from materials by Jerry Cain

JavaScript Strings

A string is an ordered collection of characters, and we can define a string using double quotations in JavaScript, i.e. `let str = "hello world"`. Each character has an index position starting from 0, i.e.

h	e	l	l	o		w	o	r	l	d
0	1	2	3	4	5	6	7	8	9	10

You can ascertain the number of characters in a string via its `length` property, i.e. `str.length`. For the string above, `str.length` would evaluate to **11**, which includes the space character " ".

Here's a circus of string methods you may find useful and/or entertaining, preferably the latter.

charAt(index)	Returns the character at the specified index of the string. You can alternatively do <code>str[index]</code> , with square brackets.
substring(start, end)	Returns the substring between indices start (inclusive) and end (exclusive).
indexOf(substr) or indexOf(substr, start)	Returns the index where the first instance of substr appears in the string, and -1 if the substring never appears. An optional 2 nd parameter start specifies the index to begin searching from.
lastIndexOf(substr)	Similar to above, but it searches backwards from the end.

For the string `str` above ("hello world"), we'd have `str.substring(1,4) => "ell"`, `str.indexOf("l") => 2`, `str.charAt(2) => "l"`, and `str.lastIndexOf("l") => 9`.

You can compare strings lexicographically—that is, by dictionary ordering—using operators such as `===`, `<`, `>`, etc. For instance, `"black" < "pink"` and `"red" !== "RED"` both evaluate to **true**.

A very useful operation is **concatenation**, which entails combining two or more strings end to end, with no intervening characters, e.g., `"hollow" + "purple"` produces the new string `"hollowpurple"`.

Reversing a String

Here's a petite function that accepts an input string `str`, and returns a string that's the reverse of the original. For instance, `reverse("stressed")` returns `"desserts"`, which I guess makes sense ☺.

```
function reverse() {
  let result = "";
  for (let i = str.length - 1; i >= 0; i--) { // iterates in reverse
    result += str.charAt(i);
  }
  return result;
}
```

Note that we build up a **new, initially empty string**, as we can't modify existing strings. This is a pattern that you'll see throughout string tasks: stitching a new string from an old one, character by character.

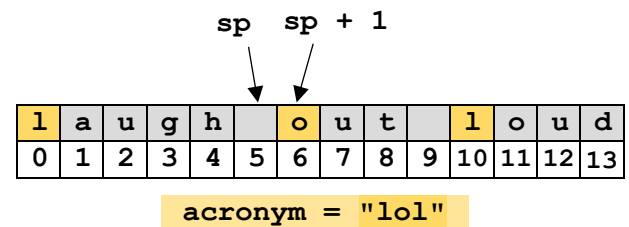
Acronyms

An *acronym* is a word formed by taking the first letter of each word in a sequence, as in:

- `acronym("National Aeronautics and Space Administration")` returns `"NASA"`
- `acronym("not in my back yard")` returns `"nimby"`
- `acronym("laugh out loud")` returns `"lol"`
- `acronym("oll korrekt")` returns `"ok"`, yes this is the origin 🍑

Our task is to implement an acronym-generating function `acronym(str)`. The first approach I can think of searches for spaces in a string, and extracts the character that immediately follows each space.

```
function acronym(str) {
  let result = str.charAt(0);
  let sp = str.indexOf(" ");
  while (sp !== -1) {
    result += str.charAt(sp + 1);
    sp = str.indexOf(" ", sp + 1);
  }
  return result;
}
```



Note that, at the beginning, we make sure to add the very first character, `str.charAt(0)`, as the only part of the acronym that does not follow a space. Furthermore, while this handles acronyms where each word is separated by a space, e.g., `"lol"`, it fails for acronyms such as `"scuba"` (*self-contained underwater breathing apparatus*), where some words are separated by hyphens instead of spaces.

Acronyms: Take II

Our second approach will correctly handle `scuba`, as well as strings with leading, trailing or multiples spaces between words, by identifying each word composed of a **sequential run of alphabet-only letters**.

- `acronym("self-contained underwater breathing apparatus")` returns `"scuba"`
- `acronym("ben! bin-yan")` returns `"bby"`
- `acronym("random-access memory")` returns `"ram"`

Key Idea: We keep track of a Boolean `inWord`. This indicates whether the character we're looking at is part of a word or not. Whenever `false` switches to `true`, we include this letter as part of the acronym!

acronym = "bby"

str	b	e	n	!		b	i	n	-	y	a	n
index	0	1	2	3	4	5	6	7	8	9	10	
inWord	T	T	T	F	F	T	T	T	F	T	T	T

First, we write a helper function that returns `true` only if the input is a valid alphabet letter. Note that, by the lexical ordering performed by comparison operators on strings, `ch >= "A" && ch <= "Z"` ensures `ch` is among letters `"A"`, `"B"`, `"C"`, ..., `"Y"`, `"Z"`, and the same applies for lowercase letters!

```
function isLetter(ch) {
  return ch.length === 1 && ((ch >= "A" && ch <= "Z") ||
    (ch >= "a" && ch <= "z"));
}
```

Next, we implement **acronym** itself! Pay attention to when **inWord** transitions from **false** to **true**.

```
function acronym(str) {
  let result = "";
  let inWord = false;
  for (let i = 0; i < str.length; i++) {
    let ch = str.charAt(i);

    if (isLetter(ch)) {
      // checks if first letter in word
      if (!inWord) result += ch;
      inWord = true;
    } else {
      inWord = false;
    }
  }

  return result;
}
```

str	b	e	n	!		b	i	n	-	y	a	n
index	0	1	2	3	4	5	6	7	8	9	10	
inWord	T	T	T	F	F	T	T	T	F	T	T	T

acronym = "bby"

Note that **inWord** is initialized to **False** to ensure we include the first letter of the starting word!

Hello World, but in JSGraphics

Revisiting your favorite library of **JSGraphics** for a moment, you can display a string graphically using the **GLabel** class. The program below etches the string "**hello, world**" on the graphics window.

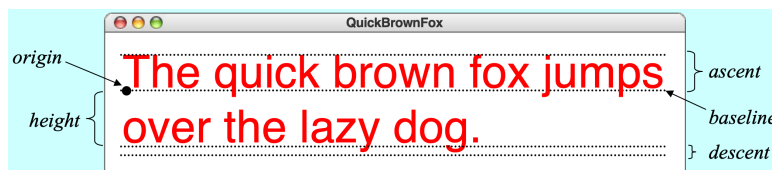
```
const GWINDOW_WIDTH = 500;
const GWINDOW_HEIGHT = 200;

function HelloWorld() {
  let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
  let label = GLabel("hello, world", 100, 75);
  label.setFont("36px Helvetica");
  label.setColor("Red");
  gw.add(label);
}
```



The function to create a label is **GLabel(text, x, y)**, and you can set the displayed font with **label.setFont(font)**, where **font** is a CSS fragment such as "**italic 36px Helvetica**", "**24px 'Times New Roman'**", and others described in our JavaScript textbook, pp. 129-131.

Note: A **GLabel**'s origin (**x, y**) differs slightly from other **GObjects**, as illustrated above, and relies on the following concepts. The **baseline** is the imaginary line on which the characters rest, and the **origin** is the point on the label begins, bottom-left



baseline at which the i.e. approximately the corner of the first line.

Vowels and Consonants

As a precursor to the **Pig Latin** program, which scintillates the final third of today's lecture, we'll compose a duet of string functions related to vowels. The first is **isEnglishVowel**, which returns **true** if the input character (i.e. a length-one string) is either an uppercase or lowercase vowel.

```
const VOWELS = "AEIOUaeiou"; // excludes "y", sorry linguists!

function isEnglishVowel(ch) {
  return ch.length === 1 && VOWELS.indexOf(ch) !== -1;
}
```

The second function relies on the first, and, given a string **word**, returns the index of the first vowel that appears in the word. If no vowel appears (e.g., "**syzygy**", one of my favorite words), we return **-1**.

```
function findFirstVowel(word) {
  for (let i = 0; i < word.length; i++) {
    if (isEnglishVowel(word.charAt(i))) {
      return i;
    }
  }
  return -1;
}
```

For instance, **findFirstVowel("apple")** returns **0**, and **findFirstVowel("scram")** returns **3**.

Pig Latin Duolingo

For this whimsical exercise, we introduce the Pig Latin language (more of a playground game, really).

There are 3 rules that enable the conversion of an English word to a Pig Latin one. Are you ready? ☺

1. If the word begins with a consonant, we move the initial consonant string (i.e. the part before the first vowel) to the end of the word, and add the suffix "**ay**". For instance, **scram** -> **amscray**.
2. If the word begins with a vowel, we simply add the suffix "**way**" to alchemize its pig Latin equivalent. For instance, **apple** -> **appleway**, and **orange** -> **orangeway**.
3. If the word contains no vowel at all, we leave the original word unchanged, e.g., **cwm** -> **cwm**.

Here's the pseudocode for a function **wordToPigLatin** which accomplishes the above translation.

```
function wordToPigLatin(word) {
  Find the first vowel in the word.
  If there are no vowels, return the original word unchanged.
  If the vowel appears in the first position, return the word concatenated with "way".
  Otherwise, divide the string into 2 parts: head and tail, before and after the vowel, respectively.
  Return the result of concatenating the tail, the head, and the string "ay".
}
```

The translation (!!) of this pseudocode into pure, beautiful, flowing JavaScript is written below.

```
function wordToPigLatin(word) {
  let vp = findFirstVowel(word);

  if (vp === -1) { // no vowel in word at all
    return word;
  } else if (vp === 0) { // word begins with a vowel
    return word + "way";
  } else { // word begins with a consonant
    let head = word.substring(0, vp); // before the first vowel
    let tail = word.substring(vp); // after and including this vowel
    return tail + head + "ay";
  }
}
```

Note that **vp** is the **index** of the first vowel, and **.substring(start, end)** excludes the **end** index.

Pig Latin: Full Sentence Translation

We expand upon **wordToPigLatin** above to translate not just individual words, but entire sentences to Pig Latin. That green Duolingo Owl would be proud 😊, even if I just lost my streak in French oops.

Our new function, **toPigLatin(str)**, should detect each English word in **str**, and translate it, e.g.,

- **toPigLatin("stout plunder lover")** returns "outstay underplay overlay"
- **toPigLatin("hello there!!")** returns "ellohay erethay!!"
- **toPigLatin("i ... am iron man")** returns "iway ... amway ironway anmay"

We define a word as a sequential run of alphabet letters, and use a similar approach to **Acronyms: Take II** to parse each word. That is, as we iterate over each character in **str**, we keep track of an **inWord** boolean that's **true** only if we're currently in a word, and **false** otherwise. We also maintain a **start** souvenir that recalls the index of the first character of the word we're currently in, and is **-1** if we're not in one. That way, **when a word ends, we'll remember where it started**, and can thus extract the word!

An example is shown below for input **"stout plunder lover"**. Notice how a new word begins when **inWord** goes from **false** to **true**, and the word ends when **inWord** transitions back to **false**.

str		s	t	o	u	t		p	l	u	n	d	e	r		l	o	v	e	r
index		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
inWord	F	T	T	T	T	T	F	T	T	T	T	T	T	T	F	T	T	T	T	T
start	-1	0	0	0	0	0	-1	6	6	6	6	6	6	6	-1	14	14	14	14	14

Here's a timeline of how each of the three words, **stout**, **plunder**, **lover**, gets parsed out of the sentence.

- When **index = 5**, **start = 0**, so **str.substr(start, index)** isolates **"stout"**.
- When **index = 13**, **start = 6**, so **str.substr(start, index)** isolates **"plunder"**.
- When the loop is over, **start = 14**, so **str.substr(start,)** lastly isolates **"lover"**.

We can use **wordToPigLatin** to translate each word 😊, which is what the pseudocode below exploits.

```

function toPigLatin(str) {
  Initialize a variable called result to hold the growing string
  for (each character position in str) {
    if (the current character is a letter) { // character is a part of a word
      if (we're not yet scanning a word) Remember the start of this word.

      } else { // is a separator character, e.g., space, punctuation
        if (we were scanning a word) {
          Extract word using the remembered start, and the current end position.
          Call wordToPigLatin to translate the word.
          Append the translated word to the result variable
        }
        Append this separator character to the result variable.
      }
    }
  }
  if (we're still scanning a word at the end) {
    Call wordToPigLatin and append the translated word to result.
  }
  Return the result string, and voila, we're done!
}

```

And lastly, here's the translation of the above pseudocode into a full-fledged JavaScript implementation.

```

function toPigLatin(str) {
  let result = "";
  let start = -1;
  for (let i = 0; i < str.length; i++) {
    let ch = str.charAt(i);

    if (isLetter(ch)) { // character is part of a word
      if (start === -1) {
        start = i;
      }
    } else { // is a separator character, e.g., space, punctuation
      if (start >= 0) {
        let currWord = str.substring(start, i);
        result += wordToPigLatin(currWord);
      }
      start = -1;
      result += ch;
    }
  }
  if (start >= 0) { // if still scanning a word at the end
    result += wordToPigLatin(str.substring(start));
  }
  return result;
}

```

Note that when **ch** is not an alphabet letter, we add it to the **result** unchanged, i.e. **result += ch**, so that all the spacing, punctuation, etc, is kept intact between the transfigured words.

Lecture 8: Arrays in JavaScript

Written by Ben Yan, from materials by Jerry Cain

Creating Arrays

An array is an ordered collection of individual data values. The values inside are called **elements**, the number of elements is called its **length**, and like strings, arrays have a **length** property you can access.

Each element is identified by its position within the array, which is referred to as its **index**. In JavaScript, like all sensical programming languages ☺, the indices begin with **0** and extend up to **length - 1**.

The simplest way to manufacture an array in JavaScript is to patiently and explicitly **list its elements separated by commas, and surrounded by square brackets** on both ends like bunny ears. For instance,

```
let coinValues = [1, 5, 10, 25, 50, 100];
```

creates an array with the standard coins in U.S. currency. I like to draw arrays as numbered boxes, e.g.,

element	1	5	10	25	50	100
index	0	1	2	3	4	5

Unlike strings, arrays can be modified in memory! Suppose that, in 2011, amid the debt-ceiling crisis, President Obama had gone through with a (real) plan to mint & issue 1-trillion-dollar coins. In this case, `coinValues.push(1e12)` would result in an updated array `[1, 5, 10, 25, 50, 100, 1e12]`.

Sum Array

This function accepts an **array** as input, adds up the elements, and returns the sum. Take note of the loop pattern expressed here—cycling through each of the array's elements in turn. It'll appear a lot ☺.

```
function sumArray(array) {
  let sum = 0;
  for (let i = 0; i < array.length; i++) {
    sum += array[i]; // accesses the element at index i
  }
  return sum;
}
```

Index Array

This function creates an array by accretion, one element a time. The returned value is an array of **n** elements equal to each of their indices, e.g., `createIndexArray(5)` returns `[0, 1, 2, 3, 4]`.

```
function createIndexArray(n) {
  let array = [];
  for (let i = 0; i < n; i++) {
    array.push(i);
  }
  return array;
}
```

Make Change

We'll write our first marquee function for arrays, **makeChange**, which given a **change** (debt) in cents, displays the number of coins of each type necessary to produce that change—using the coin values in constant arrays **COIN_VALUES** and **COIN_NAMES**. For the denominations available in U.S. currency,

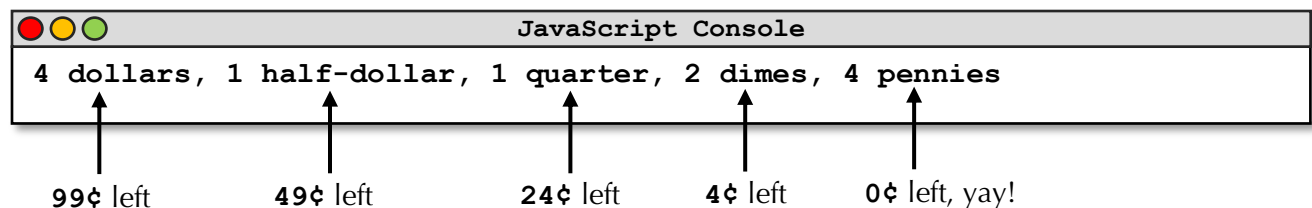
```
const COIN_VALUES = [1, 5, 10, 25, 50, 100];
const COIN_NAMES = ["penny", "nickle", "dime", "quarter",
                    "half-dollar", "dollar"];
```



We assume the currency is designed so that this (greedy) strategy always produces the correct result.

- Start with the last, most expensive element in the array (e.g., **"dollar"**), and use as many as those as possible. For instance, if **change = 499¢**, we'd use **4** dollars, leaving **99¢** in change.
- Move on to the previous element, and give as many of those as possible (e.g., for **99¢**, use **1** half-dollar), continuing this banking process until you hit rock bottom at the number of pennies.

The following facsimile of a **JavaScript Console** displays the desired output for **change = 499**.



You may assume a fully functional **createRegularPlural**, which accepts a coin name (or any **word** really), and forms its grammatically correct plural form, e.g., for **"penny"**, we'd return **"pennies"**.

First, for some remaining **change**, and a coin index **i** in the arrays above, the maximum number of those coins that can be used is given by this formula. Note with **floor**, the fractional part is discarded.

```
let n = Math.floor(change / COIN_VALUES[i]);
```

For instance, **Math.floor(499/100) => Math.floor(4.99) => 4**, if we're using dollar coins. Next, we use a non-zero number of these coins, we report it in string form, e.g., **"2 nickels"** (like the Dr. Doofenschmirtz meme). To do this, we need the coin's name, pluralizing it if more than one is used.

```
let noun = COIN_NAMES[i];
if (n > 1) noun = createRegularPlural(noun);
```

We then stringify the coin name and its quantity, separating it from other coins with a comma.

```
if (result.length > 0) result += ", "; // does not apply for 1st coin used
result += n + " " + noun;
```

Our implementation of **makeChange** is written below. We start from the highest coin, and repeat the steps above until all coins are accounted for, and so their accumulation fully compensates for **change**.

```
function makeChange(change) {
  let result = "";
  for (let i = COIN_VALUES.length - 1; i >= 0; i--) {
    let n = Math.floor(change / COIN_VALUES[i]);
    if (n > 0) {
      // put in string form how many of this coin were used
      let noun = COIN_NAMES[i];
      if (n > 1) noun = createRegularPlural(noun);

      if (result.length > 0) result += ", ";
      result += n + " " + noun; // e.g. "5 nickels"

      change -= n * COIN_VALUES[i]; // reduce remaining change
    }
  }
  return result;
}
```

Bonus: Pluralize Word

This is far from mandatory reading, as it pertains more to grammar than to arrays. But if you're curious, to implement the `createRegularPlural` function used above, we can generally follow these three rules for noun pluralization. There are lingering exceptions, of course, as English is 🙌 complicated 🙌, like *tooth* and *teeth*, *child* and *children*, *datum* and *data*. But you don't have to worry about them 😊.

- If the word ends in "s", "x", "z", "ch", or "sh", add "es" to the word. For instance, "starfish" -> "starfishes", and "matrix" -> "matrices".
- If the word ends in "y" preceded by a consonant, change the "y" to "ies". For instance, "berry" -> "berries", "sky" -> "skies", and "ivy" -> "ivies".
- In all other cases, just add an "s" 😊. For instance, "computer" -> "computers".

Here's our implementation of `createRegularPlural`, which uses a helper function defined below.

```
function createRegularPlural(word) {
  if (word.endsWith("s") || word.endsWith("x") || word.endsWith("z") ||
      word.endsWith("ch") || word.endsWith("sh")) {
    return word + "es";
  } else if (word.endsWith("y") && word.length > 1 &&
    isEnglishConsonant(word.charAt(word.length - 2))) {
    return word.substring(0, word.length - 1) + "ies";
  } else {
    return word + "s";
  }
}

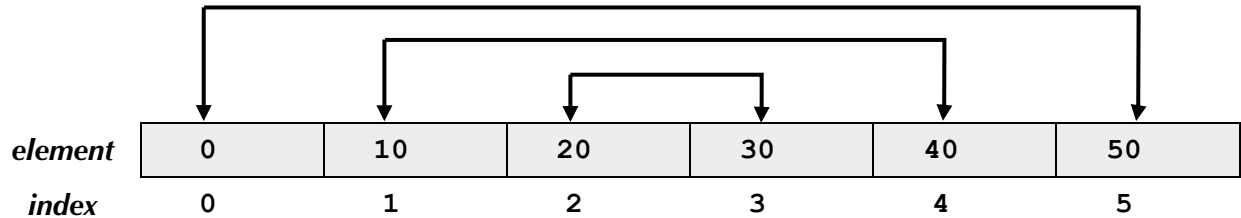
function isEnglishConsonant(ch) { // uses helper functions from Lecture 7
  return isLetter(ch) && !isEnglishVowel(ch);
}
```

Note we use the method `endsWith`, which returns `true` if the string ends with the supplied suffix.

Reverse Cursed Technique

Our last program for today **reverses the elements of an array in-place**, without creating a new one. This relies on the JavaScript principle that, when you pass an array into a function, e.g., `reverseArray`, only a **reference** to the array is actually shared, without birthing a separate copy. The effect is that **when a function changes an element of an array passed-in, that change will persist after the function returns**.

Suppose `arr = [0, 10, 20, 30, 40, 50]`, i.e. each element equals its index multiplied by 10.



Calling `reverseArray(arr)` should result in the array `arr` being updated to, no surprise, its reverse.

<i>element</i>	50	40	30	20	10	0
<i>index</i>	0	1	2	3	4	5

Note this “**reverse cursed technique**” is achieved by **swapping pairs of elements**, starting from the first and last, then the second and second last, and pinching inward, as illustrated with the arrows above.

Keeping this in mind, we’ll write a helper function that accepts an **array**, and two of its indices **lh** and **rh**, as input, and **swaps just the pair of elements at those indices**. That is, `array[lh]` becomes `array[rh]`, and `array[rh]` becomes the **original value** of `array[lh]`. To memorize this original value, before it gets overridden, we’ll need a temporary variable `tmp`.

```
function swap(array, lh, rh) {
  let tmp = array[lh];
  array[lh] = array[rh];
  array[rh] = tmp;
}
```



As a visual metaphor, try swapping the contents in two glasses of juice. You can’t do it without a third!

To implement `reverseArray`, we iterate through each pair of elements opposite across the array, and swap them, e.g., for the `arr` above, we loop over `lh = 0` and `rh = 5`, `lh = 1` and `rh = 4`, so on.

```
function reverseArray(array) {
  for (let lh = 0; lh < array.length / 2; lh++) {
    let rh = array.length - lh - 1;
    swap(array, lh, rh);
  }
}
```

Note if the length is odd, the loop ends just before we reach the element at index `array.length / 2`. But this is totally ok! The center element is unaffected by the array’s reversal, so no swaps are needed 😊.

Methods

Each of these methods has their uses cases and starring moments to shine, but the three most common operations—and the ones you should prioritize and commit to memory—are **push**, **pop**, and **shift**.

JavaScript Array Methods					
push(element, ...);					
Adds one or more elements to the end of the array, with a variable number of input parameters, e.g.,					
let arr = [];	element	1	2	4	8
arr.push(1);					
arr.push(2, 4, 8, 16);	index	0	1	2	3
pop()					
Removes and returns the last element of the array.					
shift()					
Removes and returns the first element of the array.					
unshift(element, ...);					
Adds one or more elements to the front of the array, with a variable number of input parameters.					
slice(start, finish)					
Returns a subarray beginning at the start index, and ending just before the finish index.					
splice(index, count, ...);					
Removes count elements starting at index , and optionally adds new elements. If you'd prefer to add new elements without removing any old ones, you can specify count as 0 , e.g., the method call arr.splice(i, 0, newElem) ; inserts newElem so that it now occupies index i of the array.					
concat(array, ...)					
Returns a new array that consists of all the input arrays concatenated together end-to-end, e.g.,					
let arrOne = [10, 30], arrTwo = [20, 40];					
let merged = concat(arrOne, arrTwo); // merged is array [10, 30, 20, 40]					
indexOf(element)					
Returns the first index at which element appears in the array, or -1 if not found.					
lastIndexOf(element)					
Returns the last index at which element appears in the array, or -1 if not found.					
reverse()					
Reverses the elements of the array in place. It does not return a new array but modifies the original.					
let arr = [1, 5, 10, 25, 50, 100];					
arr.reverse(); // arr is now [100, 50, 25, 10, 5, 1]					
sort()					
Sorts the elements of the array in ascending order, and modifies the original array directly.					

Keith Numbers

The last example covered today was a treatise on ✨ **Keith numbers** ✨. Such a number **num** happens to appear in a Fibonacci-like sequence with the initial terms based on **num**'s own digits, listed left to right, with each new term generating by summing the latest **n** terms, where **n** is the number of digits in **num**.

For example, **197** is a Keith number, and here's proof in the generated sequence from **[1, 9, 7]**.

1, 9, 7, 17, 33, 57, 107, 197



Note that 197 appears in its own sequence ☺, where the first 3 digits $1 + 9 + 7$ generates 17, and $9 + 7 + 17$ generates 33, and so on. We'll write a function **isKeithNumber(num)**, which returns **true** if and only if **num** satisfies the Fibonacci-esque property above, and otherwise returns **false**.

First, we write a helper function that, given an integer **num**, returns an array of its digits left-to-right.

```
function createDigitsArray(num) {
  let digits = [];
  while (num > 0) {
    let digit = num % 10;
    digits.push(digit);
    num = Math.floor(n/10);
  }
  digits.reverse();
  return digits;
}
```

1	2	3	4
---	---	---	---

[] ← digits

1	2	3
---	---	---

[4]

1	2
---	---

[4, 3]

1

[4, 3, 2]

0
[4, 3, 2, 1]

For instance, **createDigitsArray(1234)** returns the array **[1, 2, 3, 4]**. Note the digit-sum loop iterates over the digits from right-to-left, but **reverses at the very end to attain the correct ordering**.

Then, we write the actual **isKeithNumber** function, which can be implemented like this ☺.

```
function isKeithNumber(num) {
  if (n <= 0) return false;

  let partials = createDigitsArray(num); // stores the latest n terms
  while (partials[partials.length - 1] < num) {
    let sum = sumArray(partials);
    partials.push(sum); // adds the newest term
    partials.shift(); // removes the first / oldest term
  }

  return partials[partials.length - 1] === num;
}
```

From the **partials** array, initially just the digits, we generate the next number using the **sumArray** function from earlier, and add it to the array. However, since we only want to sum the latest **n** terms, we chomp off the first element in **partials**, so that it acts as a “sliding window” that contains only the most recent **n** elements, e.g., if **num = 1234**, so **n = 4**, the array starts as **[1, 2, 3, 4]**, becomes **[2, 3, 4, 10]**, then **[3, 4, 10, 19]**, and so on. Lastly, when we encounter a value at or above **num**, we exit the loop, and return **true** if we've landed exactly on **num** ✓, and **false** if we went beyond it ✗.

Lecture 9: Objects in JavaScript

Written by Ben Yan, from materials by Jerry Cain

Objects as Aggregates

JavaScript uses the term **object** in a frustratingly and hilariously imprecise way, just like, say, biweekly in the English language. Though **object** more often refers to the encapsulated data collections we'll see in upcoming lectures on object-oriented programming—for now, **object** will refer to any collection of individual data items, which other languages often call a record, dictionary (in Python), or **aggregate**.

In JavaScript, **objects** are akin to **arrays** in that **several independent pieces of data are bundled into a single data structure**. However, they differ in that the elements in an **object** can be, and often are, of differing type—and are identified by a string name (key), as opposed to by a sequence number (index).

Scrooge and Marley Ben and Jerry

Our first example for **objects** imagines keeping track of the data for employees of *Ben and Jerry* (no relation to the ice cream parlor). This is a light, cheeky modernization of *Scrooge and Marley*, which we introduced in lecture as the notorious bookkeeping company from Charles Dicken's *A Christmas Carol*.

Each employee is identified by a name, a job title, and a salary. Two examples—no surprise, Jerry & Ben—are illustrated below. There's no deeper meaning to this, of course. I don't even know what satire is.

name
Jerry Scrooge Cain
title
Professor
salary
£1000

name
Ben Cratchit Yan
title
Teaching Assistant
salary
£25



In case you're wondering why wages are in British pounds and not adjusted for inflation, I got lazy ☺. Besides, it's not like median wages have increased a whole lot in the 2000s (just ask any millennial...)

Here's how one employee would appear in code, as expressed in **JavaScript Object Notation**, or **JSON**.

```
let professor = {
  name: "Jerry Scrooge Cain",
  title: "Professor",
  salary: 1000,
}
```



Pronounced "Jason", but pls don't spell it that way lol

Under the rules of JSON, you specify an **object** simply by listing its contents as a **sequence of key-value pairs**. The key and the value are separated by a colon, the key-value pairs are separated by commas, and the entire collection is wrapped in a warm hug of curly braces. As such, here's the other employee.

```
let assistant = {
  name: "Ben Cratchit Yan",
  title: "Teaching Assistant",
  salary: 25,
};
```



Selecting Fields from an Object

Given an object, you can select an individual field / value in its memory, using an expression that includes the object name, a dot, and then the name of the corresponding key. For instance,

- `professor.name` returns the string "Jerry Scrooge Cain".
- `assistant.salary` returns the number 25, or Ben's measly income in pounds.

Field selection can also be accomplished with square brackets enclosing the key name **expressed as a string**, e.g., `professor["name"]`. This is often necessary **if the key is a variable that depends on the program's behavior**. A playful example I can think of is a `greet` function that addresses a professor by their `title`, but a student / TA by their `name`, e.g., "Hello Professor!" versus "Hello Ben!"

```
const FACULTY = ["Professor", "Chair", "Dean", "Provost", "Headmaster"];

function greet(employee) {
  let howToAddress = "name"; // key is a variable
  if (FACULTY.indexOf(employee["title"]) !== -1) {
    howToAddress = "title"; // key has been changed!
  }
  console.log("Hello " + employee[howToAddress] + "!");
}
```

Fields are assignable and modifiable. For instance, `assistant.salary *= 2` doubles Ben's salary, while `assistant.salary = 200` updates it to an entirely new amount. **Tastes like promotion!** 🍷

With an existing object, **you can also create a new key-value pair through assignment**. For instance, if we wanted to add a `catchphrase` for the employees of *Ben and Jerry*, it may look something like this:

```
professor.catchphrase = "Does that make sense to everyone? ~does T-pose~"
```

Or with the alternative, but equally hip, syntax,

```
assistant["catchphrase"] = "Jerry, we need to cook! ~makes Benzene ring~"
```

It's quite rare that you'd have to do this, but if you need to delete a field from an object for any reason, you can use the aptly named `delete` operator. For instance, if Ben decides his catchphrase is cringe (and it very much is), and needs to be erased, we could do `delete assistant["catchphrase"]`.



Nested Data Structures

Since arrays can contain values of any type, **the elements of an array can be JavaScript objects** (ooh spiky). This is far from uncommon. For example, the employees at *Ben and Jerry* can be initialized as:

```
let employees = [
  {name: "Jerry Scrooge Cain", title: "Professor", salary: 1000},
  {name: "Ben Cratchit Yan", title: "Teaching Assistant", salary: 25},
  {name: "Eugene Francisco", title: "Section Leader", salary: 10},
];
```

I've added a third person, Eugene, just for kicks. Note that despite the camouflage, **the above structure is very much still an array, and so you can index it to access each individual element**. For instance, `employees[0]` fetches the object corresponding to Jerry, so `employees[0].name` returns his name.

In general, **when you drill into nested structures, think one layer at a time**, like you're slowly peeling at an onion. Keep track of what type the current layer of nesting is, and whether to use indices/keys, e.g.

- `employees` is an array, so we use indices to go deeper, e.g., `employees[0]` for Jerry's object,

```
{name: "Jerry Scrooge Cain", title: "Professor", salary: 1000}
```

- `employees[0]` is an object, so we use keys, e.g., `employees[0].name` for this Jerry's name,

```
"Jerry Scrooge Cain"
```

- `employees[0].name` is a string, so we use indices, e.g., `employees[0].name[0]` to return his first initial, which is "J", and `employees[0].name[14]` to return his last initial, i.e. "C"

I also recommend that your variable names be clear & descriptive, to make it transparent what's stored in each variable. You can even comment the type of each variable for your own memory keeping, e.g.,

```
let profData = employees[0]; // object with 3 keys: name, title, salary
let profName = profData.name; // string, e.g., "Jerry Scrooge Cain" ☺
```

Print Payroll

The following function prints the payroll for the staff, given an `employees` array as its input argument.

```
function printPayroll(employees) {
  for (let i = 0; i < employees.length; i++) {
    let emp = employees[i];
    console.log(emp.name + " (" + emp.title + ") £" + emp.salary);
  }
}
```

For the example trio of employees listed at the top, `printPayroll(employees)` would display:

```

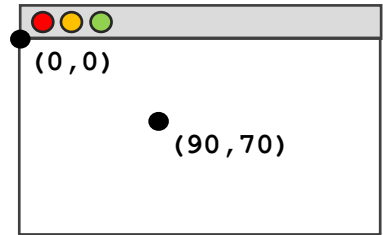
JavaScript Console
Jerry Scrooge Cain (Professor) £1000
Ben Cratchit Yan (Teaching Assistant) £25
Eugene Francisco (Section Leader) £10
```

Point Class and Factory Functions

One data aggregate useful for computer graphics captures **a point in two-dimensional space**, which is composed of an **x** and **y** component.

Points can be readily created in JavaScript with the swift ink of **JSON** object notation. A pair of examples are written below, and are also shown along with their positions in the graphics window to the right.

```
let p1 = { x: 0, y: 0 };
let p2 = { x: 90, y: 70 };
```



The **x** and **y** components of the origin point **p1** can be selected as **p1.x** and **p1.y**, respectively.

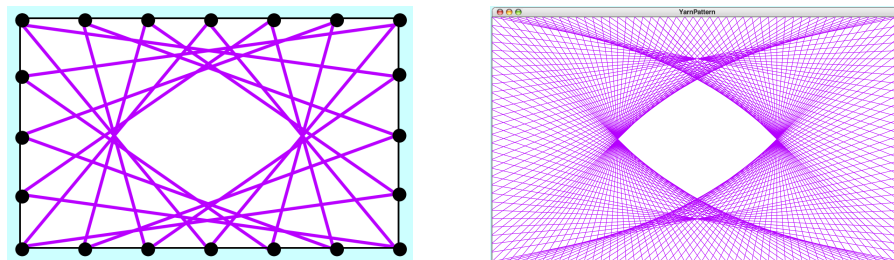
Although **JSON** notation is compact and easy to read, it's often useful to **define a function that creates a JavaScript object** from input parameters. Such functions are called **factories**. For example, this factory function below creates a point-valued object given **x** and **y** coordinates, which default to origin **(0,0)**.

```
function Point(x, y) { // style note: function name is capitalized!
  if (x === undefined) {
    x = 0;
    y = 0;
  }
  return { x: x, y: y }; // e.g., {x: 90, y: 70} for Point(90, 70)
}
```

Note: In the returned object, the **x** on the left is a key name, while the **x** on the right is a specific value.

Yarn Pattern

As an aesthetically pleasing and hopefully inspiring illustration on the use of **Point** objects in graphical applications, we present the **YarnPattern** program, using nothing but straight lines between points to simulate winding a piece of colored yarn around the pegs of a rectangular frame.



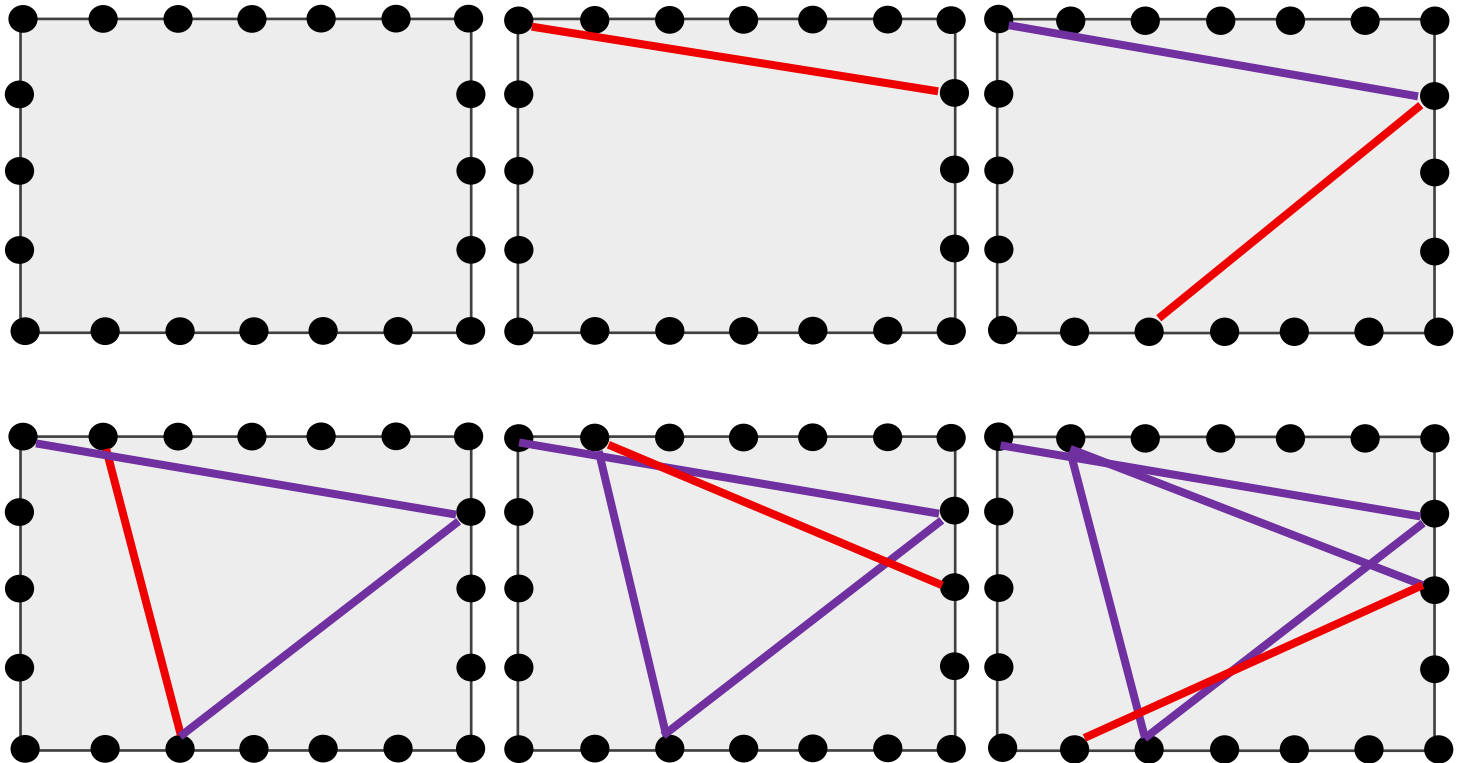
Pictures are shown above for two configurations: **(left)** 6 pegs across, 4 pegs down, each yarn jumping a “delta” of 7 pegs forward, and **(right)** 80 pegs across, 50 pegs down, and each yarn jumping 113 pegs.

We define the following constants, as shown to the right, so that we can easily modify the configuration as we wish, and dynamically create new pictures ☺. None of that Generative AI slop is needed—only our organic, home-grown JavaScript.

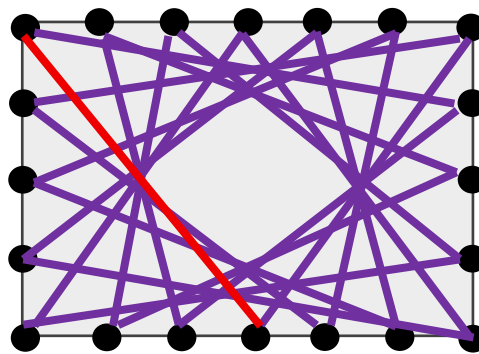
```
const GWINDOW_WIDTH = 1000;
const GWINDOW_HEIGHT = 625;
const N_ACROSS = 80;
const N_DOWN = 50;
const DELTA = 113;
```

Here's the step-by-step process we'll follow to knit this pattern. Take a deep breath ... and here we go!

- Place a set of pegs at regular intervals around the rectangular border of the graphics window.
- Tie a piece of colored yarn around the peg in the upper left corner.
- Stretch the yarn a certain number of pegs (**DELTA**) forward, in clockwise order around the frame, and wrap it around that peg. **The resulting line of yarn is shown in the top middle frame below.**
- Continue stretching forward in leaps of **DELTA** pegs, and bending the yarn into lines, until you close the loop, that is, the yarn has returned to the starting peg in the upper left corner.



The final result glimmers below. The settings are **N_ACROSS** = 6 (number of horizontal pegs **after the first**), **N_DOWN** = 4 (number of vertical pegs **after the first**), **DELTA** = 7. Note the yarn is always a royal purple in color. I just used red to artificially highlight the most recently formed line of yarn ☺.



Creating the Array of Pegs

To begin, we write a function that creates and returns an array of pegs around the perimeter of a rectangle with a specified **width** and **height** in pixels. The upper-left peg is at the origin $(0,0)$, and there are **N_ACROSS** and **N_DOWN** pegs after it in the **x** and **y** dimensions, respectively.

The **x** and **y** spacing between neighboring pegs is given by:

```
let dx = width / nAcross;
let dy = height / nDown;
```

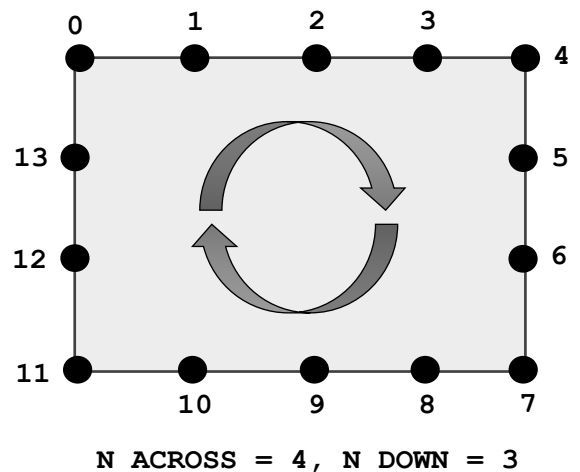
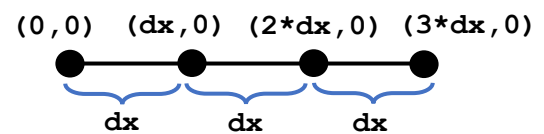
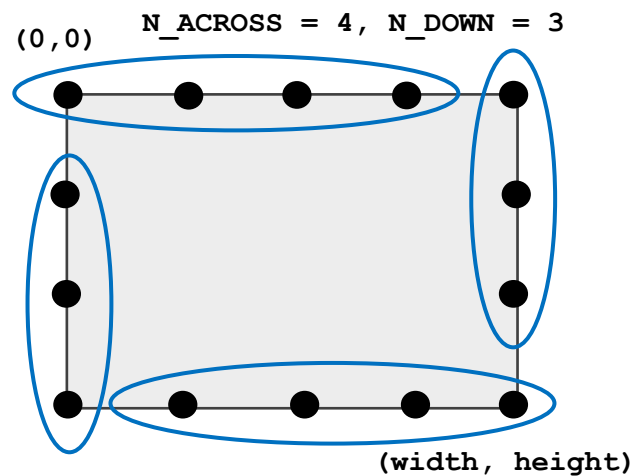
From here, we can initialize an empty array of pegs, and start adding the pegs on the top side (**y** = 0), from left to right.

```
let pegs = []
for (let i = 0; i < nAcross; i++) {
  pegs.push(Point(i * dx, 0));
}
```

Note this loop excludes the rightmost point, which features in the next loop (to add the pegs on the right side). As such, **going in clockwise order around the perimeter**, we can add all pegs in similar fashion.

```
function createPegArray(width, height, nAcross, nDown) {
  let dx = width / nAcross;
  let dy = height / nDown;
  let pegs = [];
  for (let i = 0; i < nAcross; i++) { // pegs on top side, left to right
    pegs.push(Point(i * dx, 0));
  }
  for (let i = 0; i < nDown; i++) { // pegs on right side, top to bottom
    pegs.push(Point(width, i * dy));
  }
  for (let i = nAcross; i > 0; i--) { // pegs on bottom, right to left
    pegs.push(Point(i * dx, height));
  }
  for (let i = nDown; i > 0; i--) { // pegs on left side, bottom to top
    pegs.push(Point(0, i * dy));
  }
  return pegs;
}
```

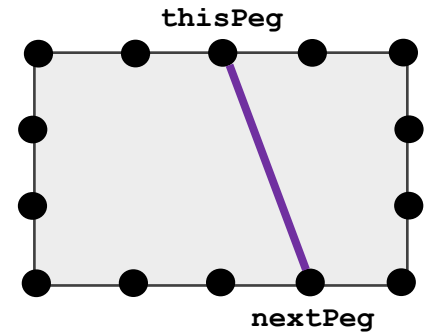
The diagram on the right shows, for the grid example above, the index of each peg in the **pegs** array. Note there are $2 * (N_ACROSS + N_DOWN)$ total pegs.



Winding Yarn Through Two Pegs

We write a helper function that creates a line of magenta purple yarn between two pegs, given their indices, **thisPeg** and **nextPeg**, in an array of **pegs**, and draws it on the graphics window **gw**.

```
function connectWithYarn(gw, pegs, thisPeg, nextPeg)
{
    let p0 = pegs[thisPeg];
    let p1 = pegs[nextPeg];
    let line = GLine(p0.x, p0.y,
                     p1.x, p1.y);
    line.setColor("Magenta");
    gw.add(line);
}
```



Recall that each element in the **pegs** array is a **Point** object, with fields **x** and **y** we can freely access

Winding Yarn Through Entire Frame 🧶

Lastly, we write the **YarnPattern** program itself, using the tools above. We start by creating the graphics window, as well as the **pegs** array, which wraps around the entire perimeter of the window.

```
let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
let pegs = createPegArray(GWINDOW_WIDTH, GWINDOW_HEIGHT,
                          N_ACROSS, N_DOWN);
```

Next, consider this snippet that moves the yarn **DELTA** pegs ahead of the current index **thisPeg**. The modulo operator **%** is used, so that indices wrap around back to **0** after exceeding **pegs.length**. For instance, if we have **14** pegs, **thisPeg = 13**, **DELTA = 5**, then **nextPeg** is $(13 + 5) \% 14 = 4$.

```
nextPeg = (thisPeg + DELTA) % pegs.length;
connectWithYarn(gw, pegs, thisPeg, nextPeg);
```

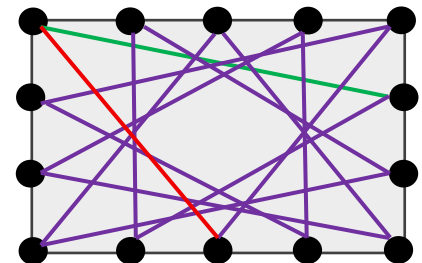
See the diagram on the
previous page for a visual!

We keep doing the above—and replacing **thisPeg** with the index of **nextPeg**, to move the yarn forward for the next iteration—until the yarn returns to the peg at index **0**. Thus, our implementation is:

```
function YarnPattern() {
    let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
    let pegs = createPegArray(GWINDOW_WIDTH,
                              GWINDOW_HEIGHT, N_ACROSS, N_DOWN);
    let thisPeg = 0;
    let nextPeg = -1; // placeholder value

    // weave through the pegs until
    // we get back to where we started
    while (thisPeg !== 0 || nextPeg === -1) {
        nextPeg = (thisPeg + DELTA) % pegs.length;
        connectWithYarn(gw, pegs, thisPeg, nextPeg);
        thisPeg = nextPeg;
    }
}
```

Start of first line!



Start of last line!

In the **while** loop, we include the condition **nextPeg === -1**, so the loop doesn't break right away!

Maps In JavaScript

One of the most important applications of JavaScript objects—besides drawing pretty pictures, of course, as art is the epitome of human expression—is using them to associate **pairs of data values**. These are still **key-value pairs**, but the connotation is the **keys are of the same type and category** (e.g., *colors*, *dorms*, *places*). **The values are generally larger and more complex**, and can even be nested maps themselves.

A classic example is a **dictionary**, and no, not Python dictionaries—but a big, physical, muscular tome like the *Oxford English Dictionary*. The keys are all words; the values are the corresponding definitions.

```
let Dictionary = {
  "syzygy": "3 or more celestial bodies aligned in a straight line",
  "defenestrate": "throw out of a window, or remove from power",
  "cabotage": "not related to sabotage, perhaps surprisingly, but the
     maritime transport of goods or passengers between 2
    places in the same country, by a carrier registered
    in a different country...I'm not making this up.",
  ... more words and definitions - and yes, these are real words ...
}
```

A more contemporary example, and one that befits the digital millennium, is the **World-Wide Web**. The keys are the URLs, and the values are the contents of the corresponding pages the URL transports you to.

Maps are naturally formed in JavaScript through the flesh of object data type. The keys are strings; the values are arbitrary JavaScript values. When you use an object as a map, it's standard to use the square-bracket notation to access values, e.g., `map[key]`, where **key** is expressed as a string variable.

- If the key is defined in the map, the selection `map[key]` returns the value.
- Otherwise, the selection doesn't crash the program, but it does return the constant **undefined**.

Map selections are assignable, so you can set the value associated with a key via `map[key] = value`. For example, if we wanted to include the "Oxford Word of the Year 2023" in our **Dictionary** above,

```
Dictionary["rizz"] = "colloquial noun for style, charm, and charisma 🤓"
```

I'm not making this up. I was studying abroad at Oxford when I learned of this, and uh culture shock.

Iterating through Keys in Map

Among the most common operations we perform with map objects is to iterate through their keys. JavaScript supports this operation through a peculiar form of the **for** loop, which is structured as:

```
for (let key in map) {
  let value = map[key];
  ... code to work with the individual key and value ...
}
```

In JavaScript, this **for** loop cycles over keys in any order it chooses. As a note of caution, when you iterate directly over the elements of an array, the syntax is **for (let elem of array)**, not **in** ☹.

Lecture 10: ASCII Character Codes

Written by Ben Yan, from materials by Jerry Cain

ASCII and Unicode Methods

Computers use numeric encodings to represent character data inside the physical memory of a machine. That is, what you see as text characters, e.g., "B", "Y", "!", "@", are really numbers in camouflage.

Each character is assigned an integer value, called a *code*, and back in the old days and medieval Silicon Valley, they weren't exactly standardized ☹️. Different computer manufacturers used their own character-to-code mappings, and sharing data across different machines was truly, truly painful.

A great renaissance was needed, and the first widespread lingua franca of character encoding was **ASCII** (*American Standard Code for Information Interexchange*). Goodness, computer scientists love their acronyms. **The ASCII standard used 128 code points, 0 to 127**. Out of them, **95 codes were for printable characters**, e.g., *alphabet letters, numeric digits, punctuation*, and **33 for control characters**, e.g., *backspace, delete, carriage return*, etc., which were critical to typewriters and early printer devices.

You can look up the table on the web, and you're not required to memorize them all—but you should know that neighboring uppercase letters have consecutive codes, which string algorithms often leverage.

Character	A	B	C	D	E	F	G	H	I	...	Y	Z
Code	65	66	67	68	69	70	71	72	73	...	89	90

That is, "A" is 65, "B" is 66, "C" is 67 🙌🙌, and so on, up to "Z", which is 90. Lowercase letters also have contiguous codes, although they start later at 97 ("a"), and skyscraper up to 122 ("z").

Character	a	b	c	d	e	f	g	h	i	...	y	z
Code	97	98	99	100	101	102	103	104	105	...	121	122

If you're curious, the character codes of numeric digits ("0" to "9") are also laid out contiguously 😊.

Character	0	1	2	3	4	5	6	7	8	9
Code	48	49	50	51	52	53	54	55	56	57

With only 128 characters, the ASCII system proved inadequate in bringing together the many alphabets in use throughout the globe, not to mention emojis 🙄. It has therefore been superseded by Unicode, which defines 172,808 characters in 175 scripts, **though the first 128 characters are identical to ASCII**.

As some fun Unicode examples, the melting face emoji, 🌡️, has character code **129760**; the Euro currency sign, €, has code **8364**; and the Chinese/Japanese character for world, 世, has code **19990**.

Unicode String Methods in JavaScript

There are two methods you'll want to remember, which enable us to translate between JavaScript characters, e.g., "B", and their Unicode numeric representations, e.g., 66. Those Unicode methods are:

String.fromCharCode (code)	Returns the one-character string whose Unicode value is code . String is hardcoded here, and refers to the string library itself.
charCodeAt (index)	Returns the Unicode value of the character at the specified index of the string calling charCodeAt .

For example, **String.fromCharCode (65)** returns "A", while **"A".charCodeAt (0)** returns 65.

The **charCodeAt** method can be used for strings of any length, but handles one index at a time. For instance, **"ABC".charCodeAt (1)** returns 66, the Unicode of the character at index 1 of "ABC".

One useful application of these methods is computing an offset or index within the alphabet. That is,

```
let offset = ch.charCodeAt(0) - "a".charCodeAt(0);
```

is an expression that, for a lowercase letter **ch**, calculates its alphabetic distance from "a". For instance, if **ch = "b"**, the offset is 1, and if **ch = "z"**, the offset is 25. If we replace "a" with upper "A", i.e.

```
let offset = ch.charCodeAt(0) - "A".charCodeAt(0);
```

we have an expression that works similarly for uppercase letters ☺. If we want to go the other way around, and find the letter that is **offset** positions after "a" in the alphabet, we can write

```
let ch = String.fromCharCode("a".charCodeAt(0) + offset);
```

Character to Upper Case

As we saw earlier, lowercase and uppercase letters have contiguously laid out character codes. One can think of them like parallel train tracks, "A" (65) to "Z" (90), and "a" (97) to "z" (122). We *capitalize* (no pun intended) on this fact to **write a function that converts a character to uppercase**.

- **charToUpperCase ("a")** returns "A".
- **charToUpperCase ("b")** returns "B".
- **charToUpperCase ("Y")** returns "Y".
- **charToUpperCase ("!")** returns "!".

A	B	C	D	E	...	Z
65	66	67	68	69	...	90



a	b	c	d	e	...	z
97	98	99	100	101	...	122



If the input character is not a lowercase letter between "a" and "z", we simply leave it as-is.

To demonstrate how we can do this for "d", we can first compute its offset from "a" using character codes. That is, **"d".charCodeAt (0) - "a".charCodeAt (0)** tells us "d" is 3 letters above "a".

Note that upper "D" is also 3 letters above upper "A", meaning we can compute the character code of "D" by the expression **"A".charCodeAt (0) + 3**, which turns out to be 68. Lastly, we convert 68 to its character equivalent via **String.fromCharCode (68)**, which yields "D", and we're done! ☺

Below is an implementation of `charToUpperCase`, using the method described above. Note we use `ch >= "a" && ch <= "z"` to check if the input is a lowercase letter ("a", "b", "c", ..., "z").

```
function charToUpperCase(ch) {
  // check if input is a lowercase alphabet letter (a, b, c, ..., z)
  if (ch >= "a" && ch <= "z") {
    let offset = ch.charCodeAt(0) - "a".charCodeAt(0);
    let upperCode = "A".charCodeAt(0) + offset;
    ch = String.fromCharCode(upperCode);
  }
  return ch; // if not a lowercase letter, character is kept as-is
}
```

Alternatively, you can use the fact that the **ASCII code of each uppercase letter is precisely 32 plus the ASCII code of the corresponding lowercase letter**. It's almost like Fahrenheit to Celsius, I think. For instance, "a" and "A" have codes 65 and 97, respectively, and "z" and "Z" have codes 90 and 122.

```
const LOWER_TO_UPPER = "A".charCodeAt(0) - "a".charCodeAt(0); // is 32

function charToUpperCase(ch) {
  if (ch >= "a" && ch <= "z") { // is lowercase letter
    ch = String.fromCharCode(ch.charCodeAt(0) + LOWER_TO_UPPER);
  }
  return ch;
}
```

String to Upper Case

Lastly, we pretend the real `toUpperCase` method doesn't exist, and write a version that returns the same result. This time, **we accept a string of any length as input**, and return an uppercase version.

- `toUpperCase("The names Bond, James Bond.")` returns `"THE NAMES BOND, JAMES BOND."`
- `toUpperCase("oh no! the band is out on the field!!")` returns `"OH NO! THE BAND IS OUT ON THE FIELD"`. This is from the legendary radio call of the 1982 Big Game.
- `toUpperCase("Harry, did you put your name in the Goblet of Fire, Dumbledore said calmly")` returns `"HARRY, DID YOU PUT YOUR NAME IN THE GOBLET OF FIRE, DUMBLEDORE SAID CALMLY"`.

Using the helper `charToUpperCase` above, the implementation of `toUpperCase` is quite gentle. We loop over each character in the input `str`, convert it to uppercase, and add it to a new running string.

```
function toUpperCase(str) {
  let result = "";
  for (let i = 0; i < str.length; i++) {
    let ch = str.charAt(i);
    result += charToUpperCase(ch);
  }
  return result;
}
```

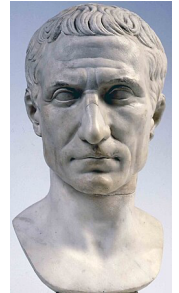
Lecture 11: Cryptography

Written by Ben Yan, from materials by Jerry Cain

Caesar Cipher

A Caesar cipher, unlike a Caesar salad or a Little Caesar's cheese pizza, may actually have been created by Roman general Julius Caesar, who was known to have used the cipher in private correspondence.

The core idea is surprisingly simple: **each letter in the plaintext**—the legible message that we're trying to disguise—is **replaced by a letter a fixed number of positions along the alphabet**. For instance, with a shift of **+3**, **"A"** would be replaced by **"D"**, **"B"** would be replaced by **"E"**, and so on, with **"W"** replaced by **"Z"**. Letters at the end of the alphabet wrap-around, e.g., **"X"** would be replaced by **"A"**, and **"Y"** by **"B"**.



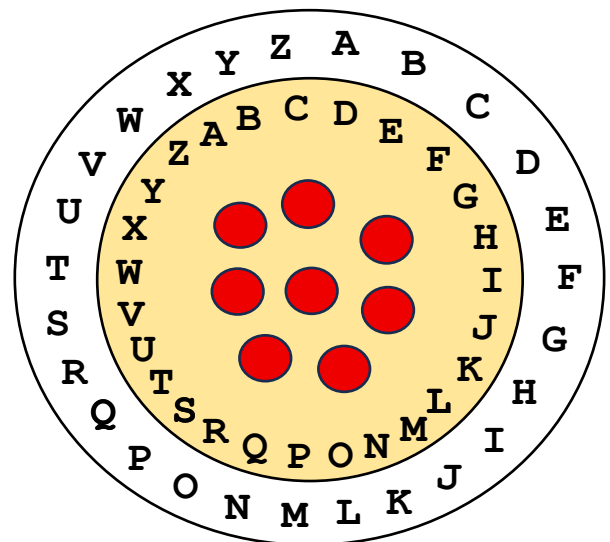
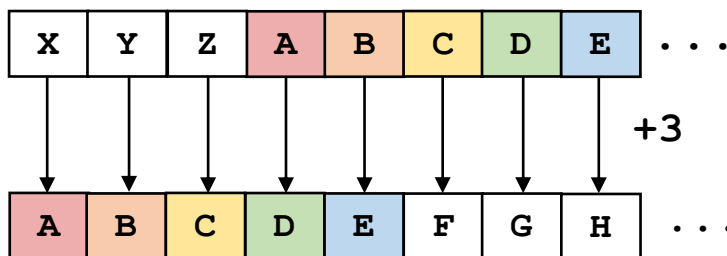
Our goal is to implement a function **caesarCipher**, which accepts a plaintext string and a shift value as input, and **returns the encrypted string that results from shifting each of the lower and uppercase letters in the input string**. Other characters, e.g., punctuation, are left as-is.

Here's a handful of examples, with famous last words from Shakespeare's *Julius Caesar* play included.

- `caesarCipher("abcd", 1)` returns `"bcde"`
- `caesarCipher("pizza", 3)` returns `"slccd"`
- `caesarCipher("Et th, Brute?", -1)` returns `"Ds sg, Aqtsd?"`
- `caesarCipher("Class of 2030", 13)` returns `"Pynff bs 2030"`



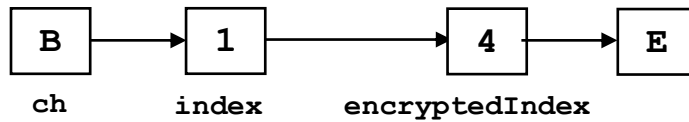
If the shift is negative, we go backward instead. For instance, if **shift = -1**, **"E"** replaced by **"D"**, and **"a"** is replaced by **"z"**. Note that a shift of **-1** is identical to a positive shift of $26 - 1 = +25$.



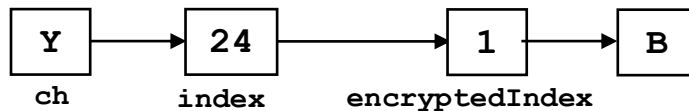
A Caesar cipher can be drawn as a **Little Caesar's pizza**, with a warm outer crust of letters, and an inner rotating circle of letters. Both run alphabetically clockwise. To encode a letter, you find it on the outer crust, and map it inward to its corresponding letter on the inner circle. The above is for a shift of **+3**.

Implement: Shift One Character

First, we'll write a helper function to **shift a single uppercase letter at a time**. To do this, we convert the character to an **index**, translate that index by the shift **key**, then convert the resulting encrypted index back into a character. Here's an example for shifting "B" by 3 positions right in the alphabet.



Here's a second example, but with wrapping around the end of the alphabet. Note that $(24 + 3) \% 26 = 1$, where the modulo operator `%` can be used to reset 26 to 0 ("A"), 27 to 1 ("B"), and so forth.



As we observed in the previous lecture, we compute a letter's index by taking the difference of its ASCII code with "A"'s ASCII code. Thus, "A" maps to index 0, "B" maps to 1, "C" maps to 2, and so on.

```
let index = ch.charCodeAt(0) - "A".charCodeAt(0);
```

From there, we can translate that **index** by the **key** shift, wrapping around the end if necessary.

```
let encryptedIndex = (index + key) % 26;
```

The ASCII code of the new letter is given by `"A".charCodeAt(0) + encryptedIndex`, and we use `String.fromCharCode` to retrieve the actual letter. Here's an implementation that achieves just that.

```
function shiftUpperCase(ch, key) {
  let code = ch.charCodeAt(0);
  let base = "A".charCodeAt(0);
  let index = code - base;
  let encryptedIndex = (index + key) % 26;
  return String.fromCharCode(base + encryptedIndex);
}
```

Next, we implement **shiftLowerCase**, which pantomimes the code above to work for lowercase letters instead. As we may expect, the only difference that we replace "A" with lowercase "a" ☺.

```
function shiftLowerCase(ch, key) {
  let code = ch.charCodeAt(0);
  let base = "a".charCodeAt(0); // uses lowercase "a" not uppercase!
  let index = code - base;
  let encryptedIndex = (index + key) % 26;
  ch = String.fromCharCode(base + encryptedIndex);
}
```

You could also 🧙 marry **shiftUpperCase** and **shiftLowerCase** into one function (as much code is repeated), or do `shiftUpperCase(ch.toUpperCase(), key).toLowerCase()` for a lowercase letter **ch**.

Implement: Shift Entire String

Next, we write the `caesarCipher` function itself. This encrypts a string `str` of arbitrary length by shifting each of its upper and lowercase letters by the `key`. To handle negative shifts, we adjust `key` to an equivalent positive shift between `+0` and `+25`, inclusive, using a bit of clever arithmetic.

```
if (key < 0) key = 26 - (-key % 26);
```

For instance, if `key = -1`, then `-key % 26 = 1 % 26 = 1`, and `26 - 1 = +25`, and indeed, shifting backward by `1` position is equivalent to shifting forward by `25` positions in the alphabet.

When in doubt on how some code works, trace through a small, tangible example for 🦋 intuition 🦋.

From here, the implementation of `caesarCipher` is, I dare say, delightfully straightforward. We iterate over each character, checking uppercase letters with `ch >= "A" && ch <= "Z"`, lowercase letters similarly with `ch >= "a" && ch <= "z"`, and shifting them using our tag team of helper functions.

```
function caesarCipher(str, key) {
  if (key < 0) key = 26 - (-key % 26); // ensures shift is positive

  let result = "";
  for (let i = 0; i < str.length; i++) { // shift each letter by key
    let ch = str.charAt(i);

    if (ch >= "A" && ch <= "Z")
      ch = shiftUpperCase(ch);

    } else if (ch >= "a" && ch <= "z")
      ch = shiftLowerCase(ch);
    }

    result += ch; // note: ch stays the same if not a letter!
  }
  return result;
}
```

And now, back in the Roman Empire, Caesar can disguise his last words to not give Brutus satisfaction.



Letter Substitution Cipher

A **cryptogram** is a puzzle in which a message is disguised by replacing each letter in the original letter with some other letter. Spooky right? The substitution pattern is consistent through the message, and exposing this pattern solves the cryptogram.

One of the most famous cryptograms was written by American writer Edgar Allan Poe, who just looks haunted (see right). In Poe's characteristically mysterious and macabre short story "*The Gold Bug*", a parchment of scrambled letters and symbols was said to unearth the location of an ancient, buried pirate treasure. Was the treasure found? Idk.



Poe's cryptogram is a wicked example of a **letter-substitution cipher**, in which **each letter in the original message is replaced by some different letter** in the coded version of that message. For this conspiracy of raven's ciphers, it's standard to represent the **key** as a string sequence of 26 letters, where **key[0]** is the letter that "A" maps to, **key[1]** is the letter that "B" is encrypted to, and so on.

As an example,

```
const KEY = "QWERTYUIOPASDFGHJKLZXCVBNM";
```

would correspond to a cipher with these letter-to-letter mappings, e.g., "A" -> "Q", "B" -> "W", etc.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Q	W	E	R	T	Y	U	I	O	P	A	S	D	F	G	H	J	K	L	Z	X	C	V	B	N	M

This is the arrangement of letters on the **QWERTY** keyboard, by the way, if you're curious ☺. Our task is to write a function **encrypt**, which accepts a string **str** and substitution pattern **key** as input, and substitutes each letter in the input string in accordance with the **key**. For instance, with the **KEY** above,

- **encrypt("ABCDEF", KEY)** returns "QWERTY"
- **encrypt("You Had Me at Hello", KEY)** returns "Ngx Iqr Dt qz Itssg". The famous quote from the 1996 rom-com *Jerry Maguire*, starring Tom Cruise and Renée Zellweger.
- **encrypt("Nevermore...Nevermore", KEY)** returns "Ftctkdgkt...Ftctkdgkt". The repeated, haunted cries of the raven in Poe's famous poem titled, no surprise, *The Raven* 🐺.

Like the Caesar cipher, we first write a helper function to **substitute a single uppercase letter**. Again, we calculate the index of the letter, and use that index to lookup the replacement letter in the **key** string.



On the next page is an implementation of **substituteUpperCase** that accomplishes exactly this.

```
function substituteUpperCase(ch, key) {
  let base = "A".charCodeAt(0);
  let code = ch.charCodeAt(0);
  let index = code - base;
  return key.charAt(index);
}
```

You won't be astonished the implementation of **substituteLowerCase** is very, very similar.

```
function substituteLowerCase(ch, key) {
  let base = "a".charCodeAt(0); // uses lowercase "a" not uppercase!
  let code = ch.charCodeAt(0);
  let index = code - base;
  return key.charAt(index);
}
```

From here, the implementation of our cryptogram-creating **encrypt** mimics that of **caesarCipher**.

```
function encrypt(str, key) {
  let result = "";
  for (let i = 0; i < str.length; i++) {
    let ch = str.charAt(i);

    if (ch >= "A" && ch <= "Z") {
      ch = substituteUpperCase(ch, key)
    } else if (ch >= "a" && ch <= "z") {
      ch = substituteLowerCase(ch, key);
    }

    result += ch;
  }
  return result;
}
```

I love you.
Ngx Egdhstzt Dt

Shut up, just shut up.
Ngx lqr Dt qz ltssg ♥

In case you're wondering how to break cryptograms, look back to no other than Poe's "*The Gold Bug*." In this story, Poe uncovers the art of **frequency analysis**: assuming the most common letters in the cryptogram likely map to the most common letters in English, i.e. **E, T, A, O, I, N, S, H, R, D, L, and U**.

For reference, here's the pirate cryptogram puzzle from the story, and a table of symbol frequencies. It may not shock you that **8** is "**E**", the most common letter in English, and the runner-up **;** is "**T**" ☺.

53†††305)) 6*;4826) 4†.) 4†) ;80
6*;48†8†60)) 85;1† (; :†*8†83(88)
5*†;46(;88*96*?;8)*†(;485);5*†
2:*†(;4956*2(5*-4)8†8*;40692
85);) 6†8) 4††;1(†9;48081;8:8†1
;48†85;4) 485†528806*81(†9;48
;(88;4(†?34;48) 4†;161;;:188;†?;

8	33	*	13	1	8	3	4
;	26	5	12	0	6	?	3
4	19	6	11	9	5	†	2
†	16	(	10	2	5	-	1
)	16	†	8	:	4	.	1

A Hitchhiker's Guide to JavaScript Syntax

Written by Ben Yan, originally for the course ❤️ CS106S: Coding for Social Good ❤️.

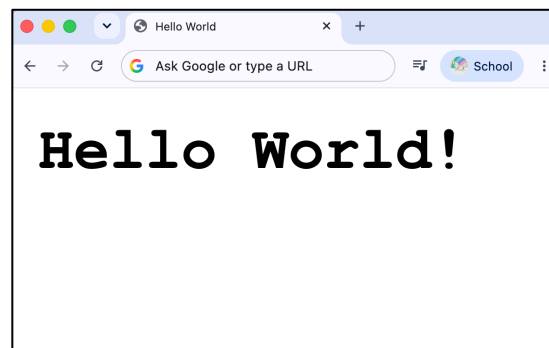
This handout records much of the basic syntax needed to write JavaScript code and programs, from print statements, to declaring and updating variables, to defining and calling functions, to conditional logic.

Topic	Pages
Printing Hello World	60
Debugging and JavaScript Console	60
Comments	60
JavaScript Arithmetic	61
Variables	62
Functions	62 – 63
Conditional Logic	63 – 65
Short Circuit Evaluation	65
More on Expression Evaluation	65 – 66
Function Declarations vs Expressions	66
Nested Functions and Closure	67

The last two sections are more esoteric compared to other programming languages, but useful for the JavaScript we teach in ✨ CS106AX ✨. **They are especially needed when writing graphics & animation programs**, where nested functions—functions defined and written in the body of other functions—completely run riot.



Syntax details for more involved data types and troublemakers—strings, arrays, objects—can be found in the corresponding lecture's notes, to avoid overwhelming you with JavaScript here ☺. You can also read about **for** and **while** loops in my *Lecture 2: JavaScript Fundamentals* notes.



Hello World

Here's a simple Hello World program in JavaScript!

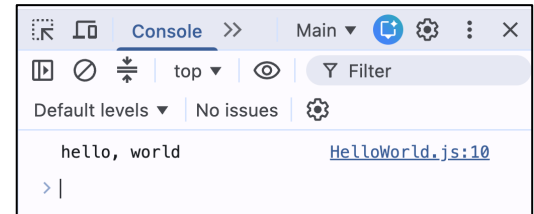
```
console.log("Hello World!");
```



You use `console.log(text)` to print to the console, as opposed to say `print(text)` in Python. If you try calling `print` in JavaScript, something very funny happens ☺ (it opens the physical printer app!)

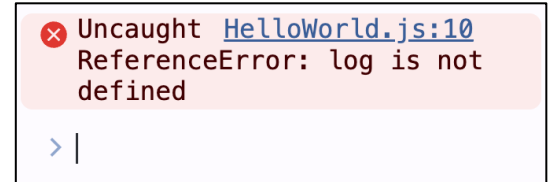
Debugging Console

To see where output from `console.log()` is delivered—unless you're using the `JSConsole.js` library to funnel it straight onto the webpage itself—you'll want to open the relevant HTML file for your JavaScript program in a browser.



Then, right-click the webpage, and click **Inspect** (as in, the notorious *Inspect Element* tool). In the resulting pop-up window, switch from the **Elements** tab to the **Console** one. Alternatively, on a Mac, you can press **Command**, then **Option**, then **J**, holding down each key while pressing the next.

This is a JavaScript console where you can type and run commands in real-time, but most importantly, **any prints from `console.log` statements in your program will show up here!** It's also where syntax and runtime errors in our code will be flagged, and you can also see the line numbers where they occur (see *right*), which is not a bad place to look 🔍.

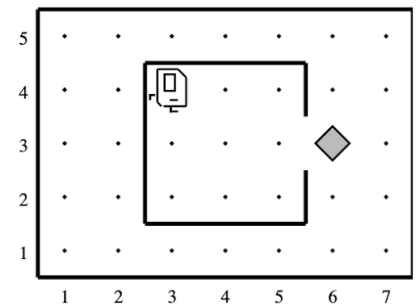


I introduced a bug in Line 10 by replacing `console.log()` with just `log()` ☺.

Comments

In JavaScript, comments begin with a `//`, and while they're not executed as lines of actual code, they can help improve readability and clarity, and signpost to the reader how your program is structured, e.g.

```
if (condition) { // robot goes left
    some code here ...
} else if (other condition) { // robot goes right
    some code here ...
} else { // robot stays on same square
    some code here ...
} don't worry about the if / else syntax here, just the commenting syntax!
```



To have comments span multiple contiguous lines, you can enclose the comment with a `/*` at the start, and a `*/` to mark the end. These are known as block or multiline comments. For instance,

```
/* Introducing myself to the computer, in a
   friendly way in case they take over the world */
console.log("Hi there, my name is Ben!");
```

When I first wrote that comment years ago, it was very much a joke, but it does seem kinda real now...

JavaScript Arithmetic

Note that math (e.g., the `+`, `-`, `*`, `/` operators) works similarly to other languages like Python and Java.

JavaScript represents all numbers as a unified `Number` type, so **the `/` operator between two integer values will perform regular division**, as opposed to integer division where the decimal part is truncated.

```
1 + 1; // => 2
10 - 4; // => 6
2 * 7; // => 14
3 / 2; // => 1.5
```

The `%` is the remainder or modulo operator, i.e. `n % k` evaluates to the remainder when `n` is divided by `k`. When `n < 0`, it's the negative of the remainder when the positive equivalent of `n` is divided by `k`.

```
4 % 2; // => 0
5 % 2; // => 1
6 % 2; // => 0
10 % 26; // => 10
```

Remainders can be useful for capturing alternating or "wrap-around" behavior in programs, e.g., time-keeping on clocks. If it's **8:00 PM** right now, then **7** hours later, it'll be **3:00 AM** (hopefully you don't stay up this late on a p-set!), as times wrap around the 12-hour clock. This could have been computed with the following JavaScript expression:



```
(8 + 7) % 12; // 8 + 7 => 15 o'clock hmm 🤔
// but clock only has 12 hours (0 - 11), so 15 % 12 => 3:00
```

The math library in JavaScript provides some useful operations as well, e.g.,

JavaScript Math Library Methods	
<code>Math.floor()</code>	
Returns	the largest integer not exceeding the input <code>x</code> , e.g., <code>Math.floor(3.14)</code> returns 3
<code>Math.round(x)</code>	
Returns	the closest integer to the input <code>x</code> , e.g., <code>Math.round(2.78)</code> returns 3
<code>Math.pow(x, y)</code>	
Returns	the value <code>x</code> raised to the <code>y</code> power, i.e. the exponent <code>x ^ y</code> .
<code>Math.sqrt(x)</code>	
Returns	the square root of <code>x</code> . Alternatively, you could call <code>Math.pow(x, 0.5)</code> .
<code>Math.max(x, y, ...)</code>	
Returns	the smallest of all the arguments. You won't be surprised there's a <code>Math.min(x, y, ...)</code> .

Lastly, having a semicolon (`;`) at the end of each statement is technically optional, as in it functionally it makes no difference in JavaScript. However, in 106AX, **our style expectations are that you include it** 😊

Variables

We use the **let** keyword to define variables of any type (whether numbers, strings, arrays, objects, etc.) The syntax in JavaScript takes the form:

```
let variableName = expression;
```

In the statement above, **=** is the assignment operator, and it assigns a value on the right side to the variable on the left side. It can also re-assign a new value to an existing variable! Here are examples.

```
let numClasses = 4;
numClasses = 5; // modifies value to 5
numClasses = numClasses - 2; // decreases to 5 - 2 => 3

let totalUnits = 12;
let avgUnitsPerClass = totalUnits / numClasses;
// the above variable equals 12 / 3 => 4
```



We also have the addition assignment (**+=**) operator, which is a pretty neat way to add a value to an existing variable. You can also do **-=** to subtract from a variable, ***=** to have it multiplied, and so forth.

```
numClasses += 1; // now back equal to 4
// same as numClasses = numClasses + 1, but shorter ☺
```

If you'd like to **increment** by precisely 1, you can do **numClasses++**; and similarly, if you'd like to **decrement** by precisely 1, you can do **numClasses--**; as a stylish shorthand.

For variables with fixed values, we can elect to use the **const** keyword instead.

```
const CS106AX_UNITS = 5; // 5-unit class, if you're an undergrad that is!
const LITERS_IN_A_GALLON = 3.785;
/* const variables should be all caps, with an underscore between words */
```

In older and legacy JavaScript code, you may encounter the keyword **var** in the place of **let** to declare variables. I used to code with **var** lol 🤖. As a general principle, avoid using **var**, it's washed and unc; if you're curious, see [here](#) for its subtle difference from **let**, which has to do with variable scope!

Functions

The general structure for defining a function in JavaScript is

```
function functionName( list of input parameters ) {
    list of statements in function body
}
```

You can optionally **return** a single value (of any type), with a statement of form **return expression**;, which communicates a value back to the caller function.

To see this in action, consider the following function, **addTwoNumbers**, which simply accepts two numbers as input, adds them together and gives back the result.

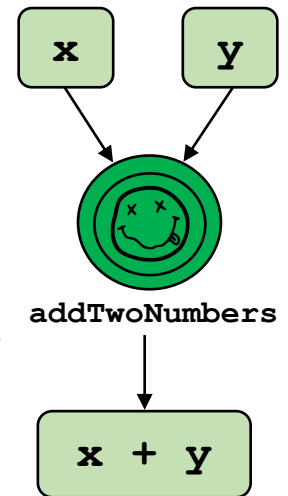


```
function addTwoNumbers(x, y) {
  let sum = x + y;
  return sum;
}
```

To call or invoke a function in JavaScript, you supply the function name, and pass in values or variable expressions for each input parameter.

```
let sum1 = addTwoNumbers(3, 4); // => 7
let sum2 = addTwoNumbers(-1, Math.PI); //=> 2.1415...

// can also compose function calls
let sum3 = addTwoNumbers(addTwoNumbers(1, 2), 3);
// this turns into addTwoNumbers(3, 3) => 6
```



If a function has a return value, when calling that function, you typically assign the result of the function call to a variable, e.g., `let result = func(inputs...)`; , which stores the value for future use ☺.

As another example, here's a JavaScript function that converts 🌡️ Celsius to Fahrenheit temperatures:

```
function celsiusToFahrenheit(celsius) {
  return 9 / 5 * celsius + 32;
}
```

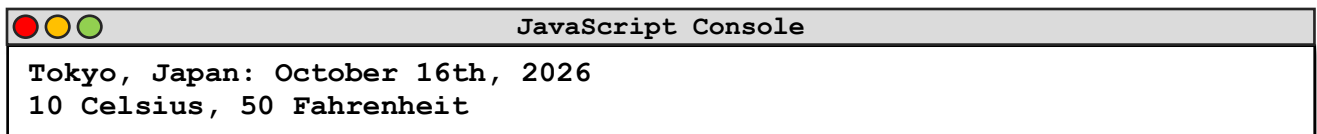
And here's a vignette of code that makes good use of the function above.

```
let TokyoC = 10;
let TokyoF = celsiusToFahrenheit(TokyoC);

console.log("Tokyo, Japan: October 16th, 2026")
console.log(TokyoC + " Celsius, " + TokyoF + " Fahrenheit");
```



The result that's printed to console is the accumulation of the string fragments above, to form:



Conditional Logic

The comparison operators `===` and `!==` check for equality and non-equality conditions, respectively

Note: We use three equal signs instead of two! It's quirky I know, but I think it's kind of beautiful ☺, and you'll get used to it. The other operators `<`, `<=`, `>=`, `>` behave as mathematically expected.

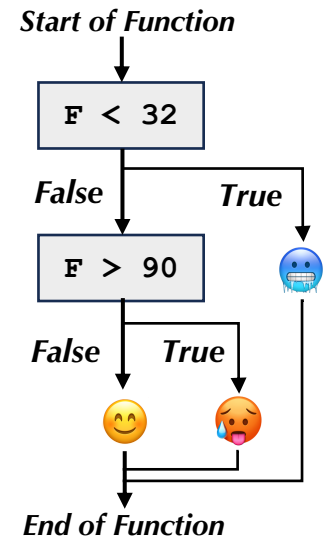
```
3 === 3 // => true
3 < Math.PI // => true
0.999 === 1 // => false
1/2 + 1/3 + 1/4 + 1/5 + 1/6 + 1/7 + 1/8 >= 2 // => false
```

Fun Fact: In general, use `===`, but JavaScript does support double equals (`==`), which is also called “*loose equals*”, as well as `!=`. Unlike `===`, *loose equals* doesn’t require the two values match in data type, and will convert one or both values into a matching type, which can lead to funny results, e.g.,

```
" " == 0; // => true, as the empty string is converted to a number, 0
" " == []; // => true, as the empty array is casted to an empty string ""
false == "0"; // => true, as both the Boolean false and string "0" are casted to number 0
```

Like most modern programming languages, JavaScript supports **if**, **else if**, and **else** conditional statements, for a program to behave differently or **branch under varying conditions**. To see these in action, consider a 🐼 **Goldilocks** function that takes, as input, the temperature outside, and judges it.

```
function Goldilocks(fahrenheit) {
  if (fahrenheit < 32) {
    console.log("It's too cold outside!");
  }
  // checks this only if above condition fails
  else if (fahrenheit > 90) {
    console.log("It's too warm outside!");
  }
  // only when both conditions above fail
  else{
    console.log("The weather is just right.");
  }
}
```



Note this function will print that it's "**just right**" only when the Fahrenheit temperature is between **32** and **90**, i.e. it's **at least 32** degrees AND it's **at most 90** degrees. Otherwise, it's too warm or cold ☹️.

Say instead we want a function that **returns the Boolean true** if the temperature is "just right" 😊, and **false** if it's too hot 🥵 or too cold 🥶. For this, in JavaScript, we can leverage the `&&` operator for logical AND, the `||` operator for the logical OR, and the unary operator `!` for the logical NOT.

```
function isGoldilocks(fahrenheit) {
  return fahrenheit >= 32 && fahrenheit <= 90;
}
```

Here's a function, **isNotGoldilocks**, which—surprise—returns the exact opposite.

```
function isNotGoldilocks(fahrenheit) {
  return fahrenheit < 32 || fahrenheit > 90;
  // true when weather too cold OR too hot, false when just right
}
```

Alternatively, you could call **isGoldilocks**, and flip the result with the logical NOT operator, i.e. `!`.

```
function isNotGoldilocks(fahrenheit) {
  return !isGoldilocks(fahrenheit);
  // with the ! operator, true becomes false, and false becomes true
}
```

Short Circuit Evaluation

JavaScript evaluates the `&&` and `||` operators using a time-saving strategy called **short-circuit mode**, in which it evaluates the right operand only if it needs to do so. For instance, in the Boolean expression

$$n \neq 0 \ \&\& \ x \% n === 0$$

if `n` equals `0`, the right operand of `&&` is not evaluated at all, because `n !== 0` is **false**. In general,

$$\text{false} \ \&\& \ \text{anything}$$

is always **false**, meaning the rest of the expression no longer matters. Similarly, the expression

$$\text{true} \ || \ \text{anything}$$

is always **true**, so the rest of expression is irrelevant to the result, and thus is not actually evaluated 😊.

An advantage of short-circuit evaluation is that you can strategically use `&&` and `||` to prevent runtime errors. If `n` were `0` in the earlier example, evaluating `x \% n` would crash due to division by zero ☹️.

More on Expression Evaluation

JavaScript obeys the PEMDAS rule—perhaps reminiscent of one’s grade school math courses—for arithmetic **order of operations**. That is, in order of highest to lowest precedence, we evaluate

- **Parentheses** `()`
- then **Exponents** `**`
- then **Multiplication** `*`, **Division** `/`, and **Remainder** `%` operators (*tiebreaker, left to right*)
- then **Addition** `+` and **Subtraction** `-` operators (*tiebreaker, left to right*)



As a rule of thumb, we evaluate **arithmetic operators first** (`+`, `-`, `/`, `*`), then **comparison operators** (`<`, `<=`, `>`, `>=`, `===`, `!==`), then **logical / Boolean operators** (`&&`, `||`). One niche exception is logical NOT (`!`), which has high priority, and is evaluated before arithmetic operators but not parentheses.

For example, consider the expression `2025 > 2020 + 6 || 3 \% 3 === 3`, which evaluates to

Arithmetic operators	→	<code>2025 > 2020 + 6 3 \% 3 === 3</code>
Comparison operators	→	<code>2025 > 2026 0 === 3</code>
Logical operators	→	<code>false false</code>
		false



Let’s also consider the **evaluation of string expressions** 😊. Note that with JavaScript strings, **`+` is the concatenation operator, which combines strings end-to-end with no intervening characters**, e.g., `"blue" + "red"` evaluates to `"bluered"`. A string and a number be concatenated into a string as well, e.g., `"class" + 2030` evaluates to `"class2030"`, and `670 + "Lomita"` to `"670Lomita"`.

When concatenating multiple values in a row, note order of operations rules, keep track of type (string or number), and evaluate two at a time. For instance, the expression `"messier" + 2 * 3 + 7` evaluates to `"messier" + 6 + 7`, then `"messier6" + 7`, and lastly, `"messier67"`. Meanwhile, `2 * 3 + 7 + "messier"` evaluates to `6 + 7 + "messier"`, then `13 + "messier"`, and lastly, `"13messier"`. Spooky, but it’s different due to left-to-right rules and `+` as addition vs. concatenation!

Functions Declarations vs. Expressions

We've talked about two main ways to create a function in JavaScript, which are **declarations** and **expressions**. A **declaration** simply uses the **function** keyword and specifies a function name, e.g.,

```
function greet(name) {
    console.log("Hello " + name + "!");
}
```



The second way, which we witness more often in graphical programs (especially ones that involve interactivity / animation), creates a **function expression**, and assigns it to a variable that can be called.

```
let greet = function (name) {
    console.log("Hello " + name + "!");
}
```



A key functionality difference is that **declared** functions are fully loaded before any code has even run. Thus, you can call / refer to them even before the line of code they're defined in, e.g., the below is ok!

```
drawCircle(); // executes the (pre-loaded) function below!
```

```
function drawCircle() {
    ... code that draws a randomly colored circle on the canvas ...
}
```

Meanwhile, the code below, involving a function **expression**, would crash ☹. This is because function expressions are defined only when we reach their line of code upon executing the program line-by-line.

```
drawCircle(); // Reference Error: Can't access before initialization
```

```
let drawCircle = function() {
    ... same code that draws a randomly colored circle on the canvas ...
}
```

Despite the runtime risks lol, you may stylistically prefer a function expression with **let** to emphasize that functions, like any other data type in JavaScript (e.g., numbers, strings, arrays), are stored as data variables, and thus, they be passed as parameters into other functions, e.g., in our Draw Dots program,

```
let drawDot = function(e) {
    ... (Lecture 4) code that draws a filled dot centered at where the user clicked ...
}

gw.addEventListener("click", drawDot); // passes function as variable in!
```

You can also pass a function as an input parameter without explicitly assigning it a variable name, e.g., below. A caveat is that you can't refer to this function later in your code, as it's not stored in a variable.

```
gw.addEventListener("click", function(e) {
    ... same code that draws a filled dot centered at where the user clicked ...
});
```

Nested Functions and Closures

JS supports nested functions, i.e. **functions defined in the body of another function**. Here's an example that may resonate with you, with Stanford, the GovCo area, and two of its dorms: Potter and Schiff ☺.

```
function Stanford() {
  let Farrillaga = 341; /* address is 341 Galvez St */
  let Tresidder = 459; /* address is 459 Lagunita Drive */
  ... code here can access Farrillaga and Tresidder ...

  function GovCo() {
    let Ricker = 238; /* address is 238 Santa Teresa St */
    ... code here can access Ricker, Farrillaga, and Tresidder ...

    function Potter() {
      let Cats = 3;
      ... code here can access Cats, Ricker, Farrillaga, and Tresidder ...
    }

    function Schiff() {
      let Dogs = 4;
      ... code here can access Dogs, Ricker, Farrillaga, and Tresidder...
    }
  }
}
```



The **closure** of a function is defined as a **record storing a function together with all the variables it had access to at the time of creation**. One can think of closure metaphorically as an ID keycard that retains access to these variables, even when the outer function has stopped executing. In JavaScript, **the scope rules are that each function has access to its local variables, but also upward access to local variables in its outer functions**. However, the reverse is not true: an outer function cannot access its inner functions' own variables, creating a membrane seal of privacy, e.g., a **Govco resident*** not in **Potter** or **Schiff** can't access their **Cats** or **Dogs**, whereas a **Potter resident*** can access **Cats**, but also has keycard access to variables **Ricker**, **Farrillaga**, and **Tresidder** in the nesting dolls of its outer functions.



***resident** = line of code residing in a function (my metaphor ☺)

