# Lecture 07: Objects and Visualization

COSC 225: Algorithms and Visualization

Spring, 2023

## Outline

1. JavaScript Events
2. Activity: Draw Dots
3. Objects in JavaScript
4. Graphs

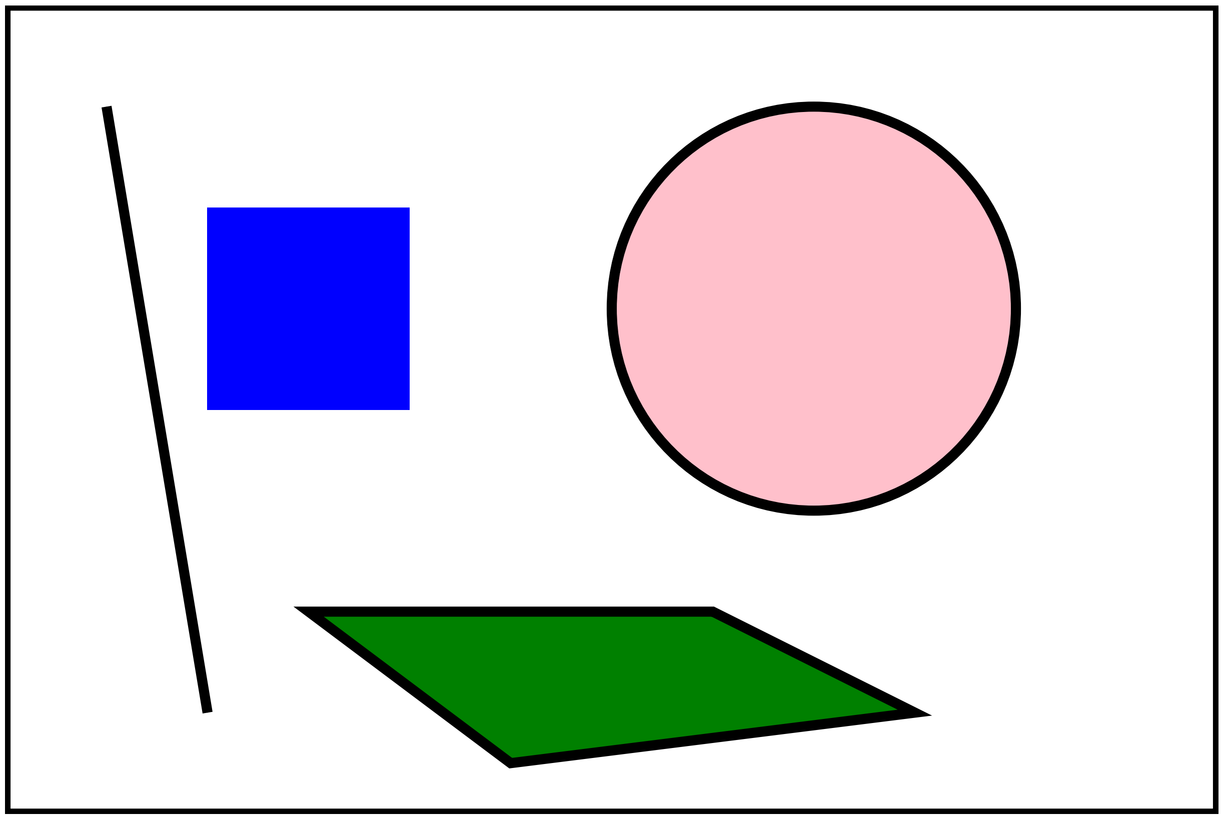
## Last Time: SVG

**S**calable **V**ector **G**raphics

- format for representing graphical objects

```
<svg width="600" height="400" xmlns="http://www.w3.org/2000/svg">
  <rect width="100%" height="100%" fill="white"
    stroke="black" stroke-width="5"/>
  <rect x="100" y="100" width="100" height="100"
    fill="blue" stroke="none"/>
  <circle cx="400" cy="150" r="100" stroke="black"
    stroke-width="5" fill="pink"/>
  <polygon points="150 300 350 300 450 350 250 375"
    stroke="black" stroke-width="5" fill="green"/>
  <line x1="50" y1="50" x2="100" y2="350"
    stroke="black" stroke-width="5"/>
</svg>
```

## Output



# Today: Interacting with SVG

- So far: using JS to add elements to page
  - Color Grid
  - Cellular Automata
- Same techniques can be applied to SVG

namespace

string

"www.w3.org/…"

  - create elements

```
let circle = document.createElementNS(ns,
'circle');
```

namespace (not needed after creation)

  - set attributes

```
circle.setAttributeNS(null, "fill", "pink");
```

attribute

value

  - add elements

```
svg.appendChild(circle)
```

some svg element

What about interactions **in response** to user?

## JavaScript Events

Goal: call a method (or methods) when user **interacts** with elements on the page

Examples:

- `click` on an element
- `mouseover` an element
- `mousemove`
- typing on keyboard

These are all **Events** in JS!

## Adding Events Listeners

Call a method when an element is clicked:

```
// get the element you want to add the listener to
const box = document.querySelector("#dot-box");

// add the listener
box.addEventListener("click", drawDot);
```

some element

- "click" is the name of the event we are listening
- drawDot is the method that will get called when event occurs
  - method gets passed an Event object
  - contains info about the Event

drawDot (e)

Event object

## Event Attributes

If `e` is a (mouse) `Event`, such as `click`:

- `e.clientX` = x-coordinate of where the event occured
- `e.clientY` = y-coordinate of where the event occured
- `e.target` = element that “heard” the event

## Demo: Click!

## Activity

Draw dots on your SVG!

## Homework 05

Draw other stuff as well!

- basic: just draw lines
- extra credit: draw more!

## Objects in JavaScript

## What are Objects?

Collection of

- attributes and associated values
- methods

**Example** `dot` class

- attributes:
  - `cx` x position of center
  - `cy` y position of center
- methods:
  - `updateLocation(cx, cy)` moves dot to a new location

## Object Constructors

In JS, object types can be defined by defining a **constructor**

- function that creates the object
- keyword `this` defines attributes and methods

By convention, constructor names are Capitalized:

```
function Dot(cx, cy) {
    this.cx = cx;
    this.cy = cy;
    this.circle = document.createElementNS(ns, 'circle');
    this.circle.setAttributeNS(null, 'cx', this.cx);
    this.circle.setAttributeNS(null, 'cy', this.cy);
    this.circle.setAttributeNS(null, 'class', 'dot');
    svg.appendChild(this.circle);
}
```

field (instance variable)

## To make individual dots

```
let someDot = new Dot(100,100);
let anotherDot = new Dot(200,200);

// refer to Dot fields
let x = someDot.cx; // x stores value 100
```

```
let e = someDot.circle
    e.setAttributeNS(null, "cx", 100)
```

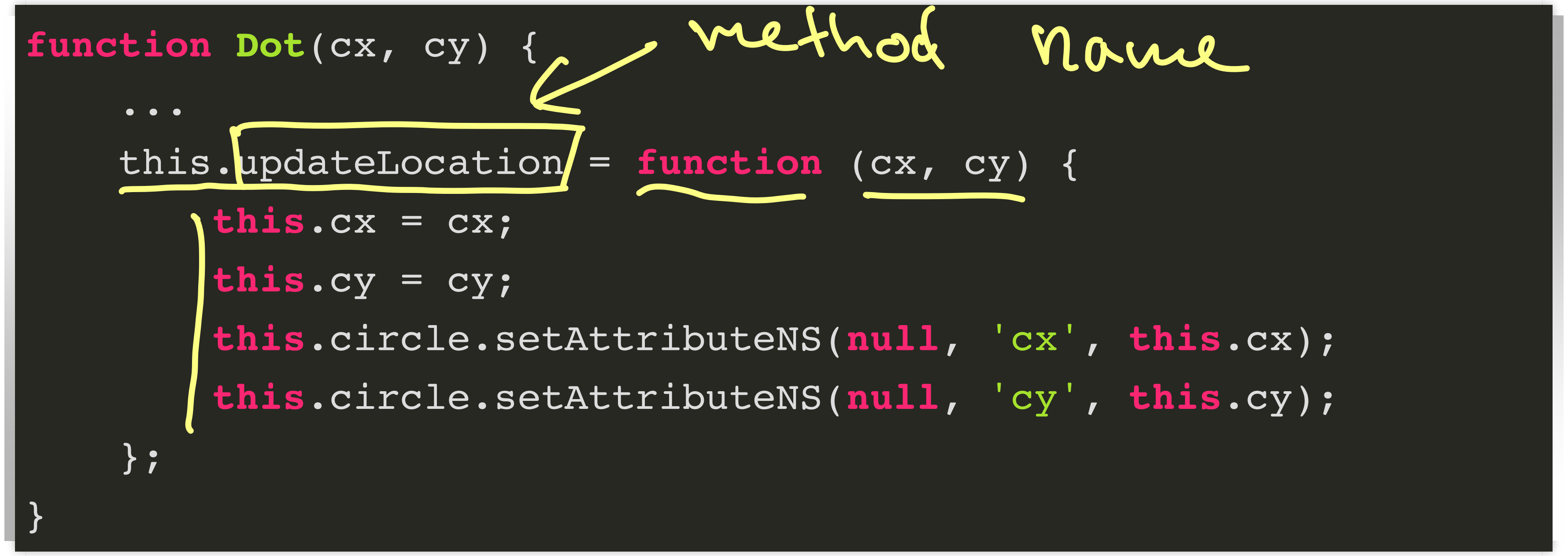
## Now to make some dots...

```
dots = []; // an array of dots

function makeDots() {
    for(let i = 0; i < 10; i++) {
        let x = Math.floor(600 * Math.random());
        let y = Math.floor(400 * Math.random());
        dots.push(new Dot(x, y));
    }
}
```

## Defining Methods

You can include method definitions in the constructor as well!



```
function Dot(cx, cy) {
    ...
    this.updateLocation = function (cx, cy) {
        this.cx = cx;
        this.cy = cy;
        this.circle.setAttributeNS(null, 'cx', this.cx);
        this.circle.setAttributeNS(null, 'cy', this.cy);
    };
}
```

let dot = dots[3];
dot.updateLocation(100, 100)

## Now we can move dots around

```
dots = [];

//...create dots...

function moveDots() {
    for(let i = 0; i < 10; i++) {
        let x = Math.floor(600 * Math.random());
        let y = Math.floor(400 * Math.random());
        dots[i].updateLocation(x, y);
    }

}
```

## Dots Demo

## Graphs

## Graphs

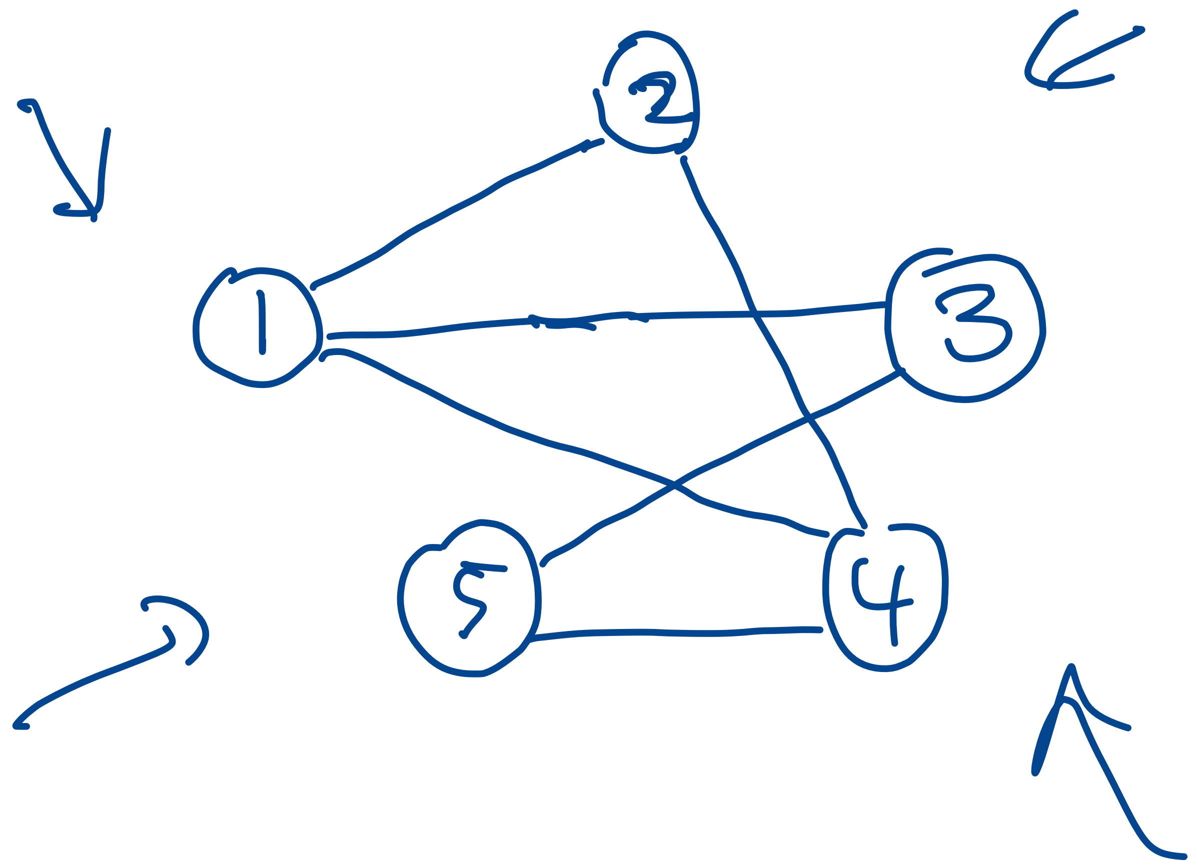
Mathematical abstraction of *networks*

- set $V$ of $\boxed{\textbf{vertices}}$ a.k.a. **nodes**
- set $E$ of **edges**
  - each edge $e \in E$ is a *pair* of nodes

If $\underline{(u, v) \in E}$, we say $u$ and $v$ are **neighbors**

## Example

- $V = \{1, 2, 3, 4, 5\}$
- $E = \{(1,2), (1,3), (1,4), (2,4), (3,5), (4,5)\}$



## Representing Graphs

Adjacency list representation

- list (e.g., array) of vertices
- for each vertex, store a list of its neighbors

**Example**

- $V = \{1, 2, 3, 4, 5\}$
- $E = \{(1, 2), (1, 3), (1, 4), (2, 4), (3, 5), (4, 5)\}$

1 : 2 3 4

2 : 1 4

3 : 1 5

4 : 1 2 5

5 : 3 4

# Graphs as Objects

**Question.** Suppose we want to write a JavaScript program to represent and manipulate graphs. What **types** of objects might we want to represent?

- Vertex
    - neighbors
    - id
- Edge
    - vertices it connects
- Graph : vertices, edges?

## What fields/ops should `Graph` have?

- add/remove vertices
- add/remove edge
- find vertex

## What fields/ops should `Vertex` have?

- change neighbor

## Anything else?

## Designing JavaScript Graphs

My Goals:

- represent and manipulate graphs
- **visualize** graphs

## Question

What additional information/functionality should our `Graphs` (and related objects) have to support visualization and user interaction?

## Graph Demo

## My Design

- `Graph` object that stores vertices, edges, visualizer
- `Vertex` stores id, adjacency list, location, graph
- `Edge` stores endpoints, id
- `GraphVisualizer` stores graph, svg element, text field
    - handles all drawing and user interactions

## Design Principles

- encapsulation: break functionality into small, logically independent pieces
- different functionality $\implies$ different objects
  - separate representation from presentation/interaction

## Next Time

Visualizing Graph Algorithms!

- simple animations