D3 Tutorial

Interactions

Edit by Jiayi Xu and Han-Wei Shen, The Ohio State University

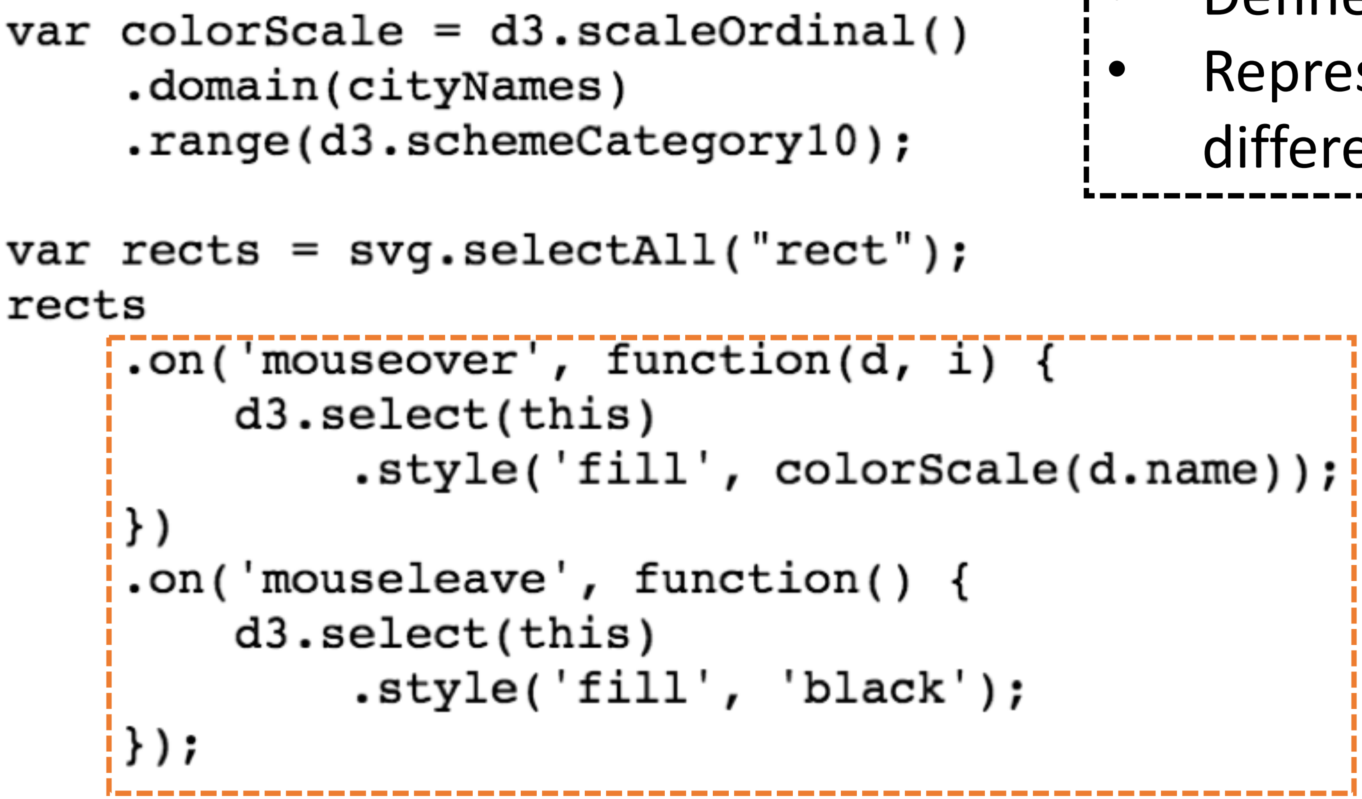
Interactions

- It is often required to interact with our visualizations, e.g. hovering, zooming, clicking etc., to change the appearance of the visual elements or drill drown information
- Topics
  - Mouse events
  - Drag
  - Zoom and pan
  - Brush

Mouse Events

- Mouse events like *click*, *mousedown*, *mouseenter*, *mouseleave*, *mouseover* etc. are vey common in UI interaction
- *selection*.on(*EventType*, *listener*)
  - Register an event listener to a selection
  - *EventType* is the name (string) of a event type, e.g., *click*, *mouseover*, etc.
    - Any DOM event type supported by your browser may be used (not only mouse events)
      - Event list: https://developer.mozilla.org/en-US/docs/Web/Events#Standard_events
  - When a specified event is triggered, the *listener* function will be invoked

Mouse Events – Populations of Cities

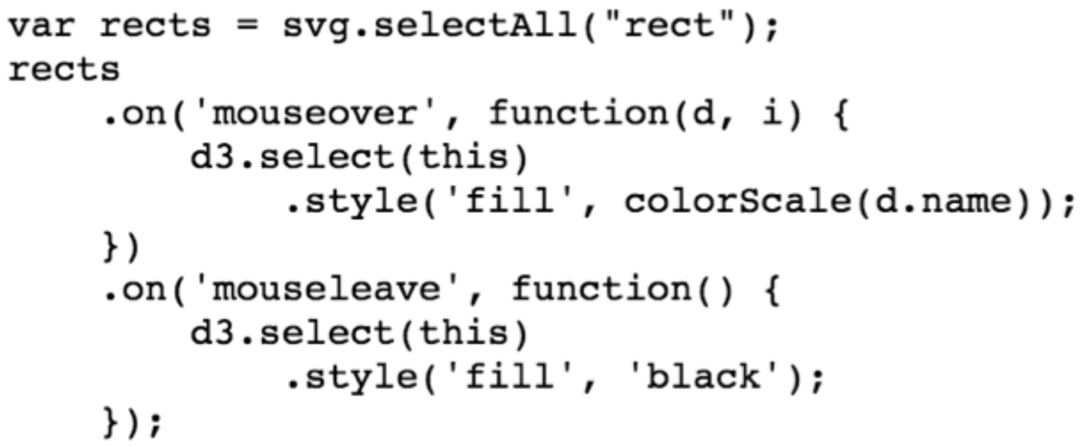


```
var colorScale = d3.scaleOrdinal()
     .domain(cityNames)
     .range(d3.schemeCategory10);

var rects = svg.selectAll("rect");
rects
     .on('mouseover', function(d, i) {
          d3.select(this)
               .style('fill', colorScale(d.name));
     })
     .on('mouseleave', function() {
          d3.select(this)
               .style('fill', 'black');
     });
```

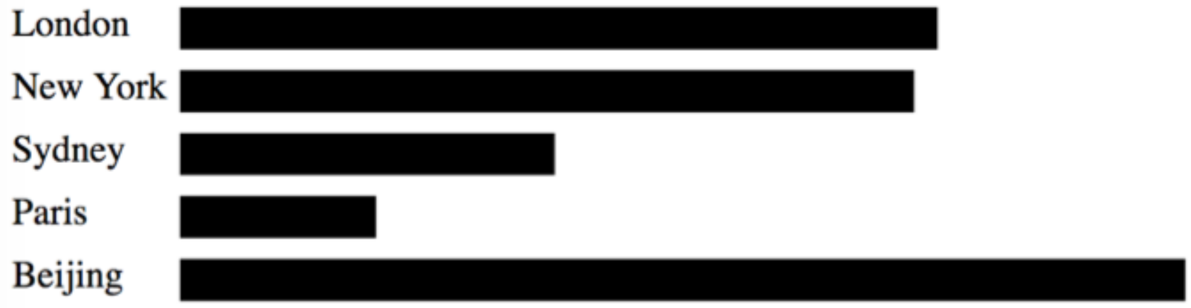
- Define a color scale
- Represent cities by different colors

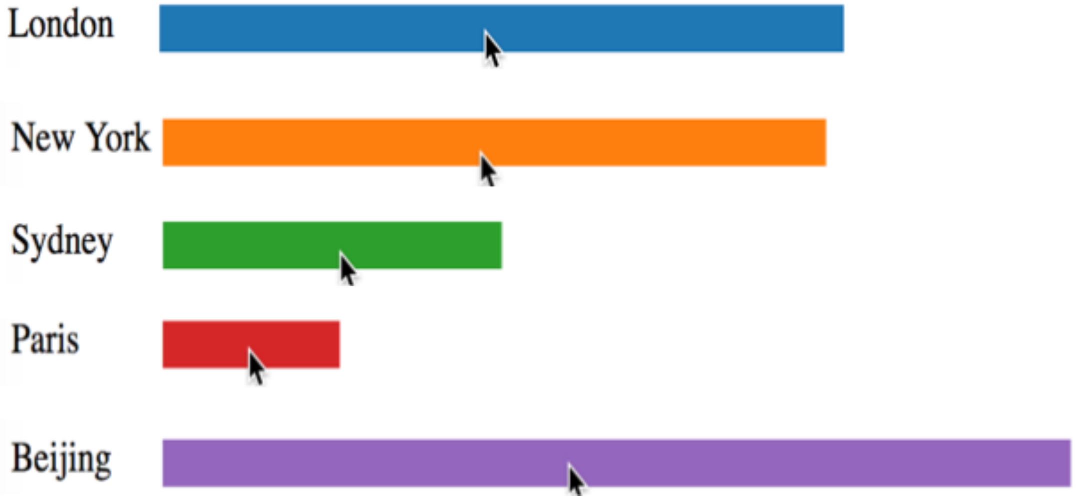
- When mouse hovers cities' bars, we use different colors to highlight bars.
- The variable *this* stores the related *rect* element
- When mouse leaves bars, we repaint bars in black.

Mouse Events – Populations of Cities



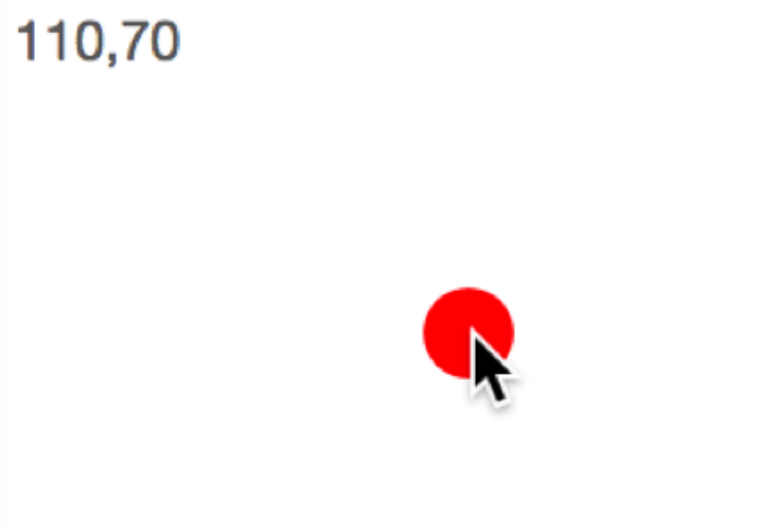
```
var rects = svg.selectAll("rect");
rects
    .on('mouseover', function(d, i) {
        d3.select(this)
            .style('fill', colorScale(d.name));
    })
    .on('mouseleave', function() {
        d3.select(this)
            .style('fill', 'black');
    });
```



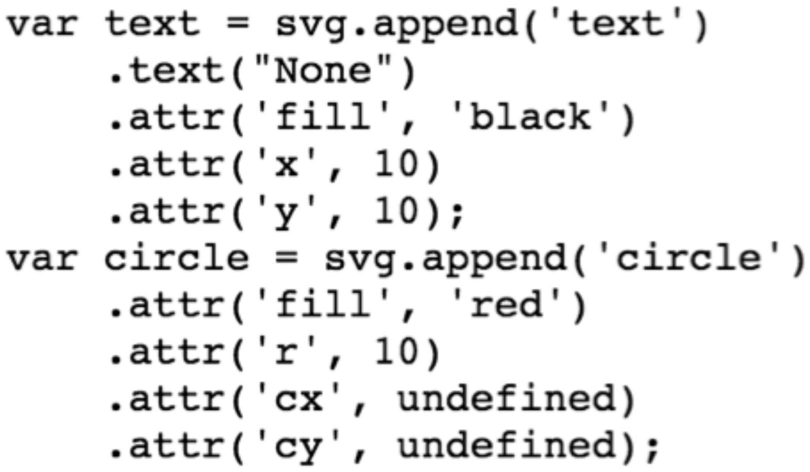


Mouse Events – d3.mouse(*container*)

- d3.mouse(*container*)
  - Returns the x and y coordinates of the current event relative to the specified container
    - The container is a DOM element such as a *svg* or *g* element
- Example
  - When mouse moves, show the position of the mouse on the screen



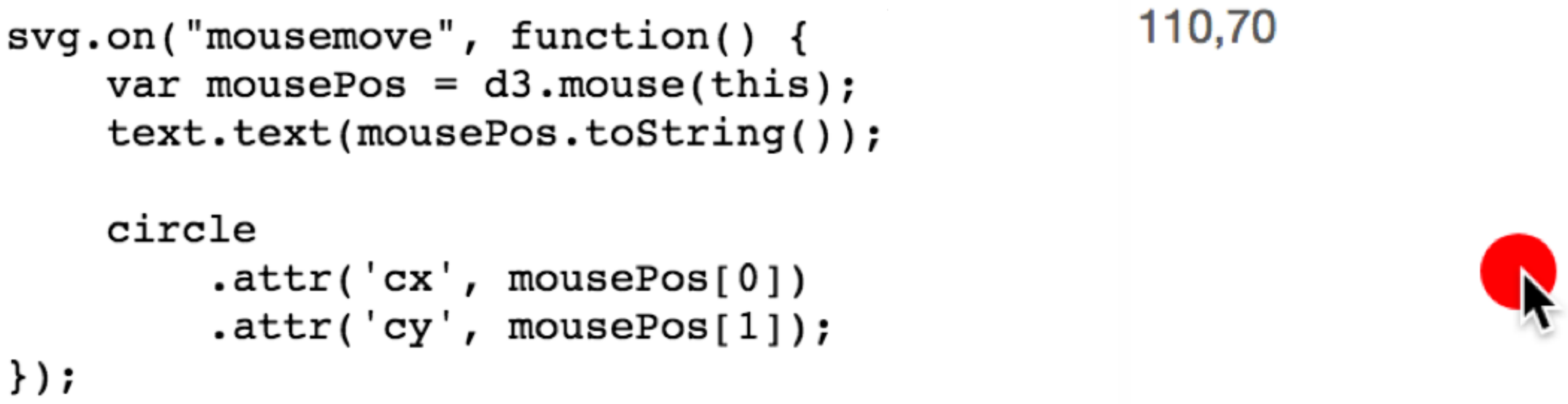
```
110,70
```



```
var text = svg.append('text')
    .text("None")
    .attr('fill', 'black')
    .attr('x', 10)
    .attr('y', 10);
var circle = svg.append('circle')
    .attr('fill', 'red')
    .attr('r', 10)
    .attr('cx', undefined)
    .attr('cy', undefined);
```

- First, create the *text* and *circle* tag

Mouse Events – d3.mouse(*container*)



```
svg.on("mousemove", function() {
    var mousePos = d3.mouse(this);
    text.text(mousePos.toString());

    circle
        .attr('cx', mousePos[0])
        .attr('cy', mousePos[1]);
});
```

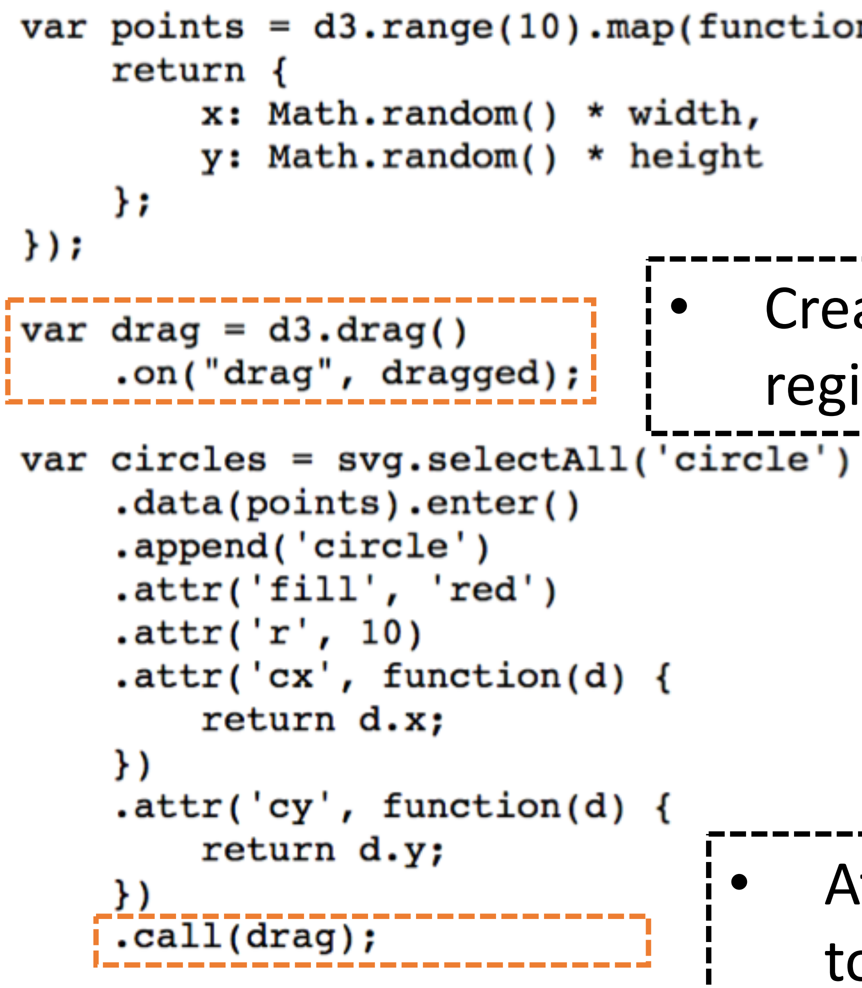
```
110,70
```

- d3.mouse(*this*)
  - Returns the *x* and *y* coordinates (as an array [*x*, *y*]) of the current event (*mousemove*) relative to the specified container (*svg*)
    - *this* represents the *svg* element
  - Equivalent to d3.mouse(svg.node())
    - *selection*.node() returns the DOM element of the selection (here is *svg*)

Drag Behavior

- Drag-and-drop is a popular and easy-to-learn pointing gesture
  - move the pointer to an object
  - press and hold to grab it
  - “drag” the object to a new location
  - release to “drop”
- D3’s drag behavior provides a convenient but flexible abstraction for enabling drag-and-drop interaction on selections

Drag Behavior – d3.drag()



```
var points = d3.range(10).map(function() {
    return {
        x: Math.random() * width,
        y: Math.random() * height
    };
});

var drag = d3.drag()
    .on("drag", dragged);

var circles = svg.selectAll('circle')
    .data(points).enter()
    .append('circle')
    .attr('fill', 'red')
    .attr('r', 10)
    .attr('cx', function(d) {
        return d.x;
    })
    .attr('cy', function(d) {
        return d.y;
    })
    .call(drag);
```

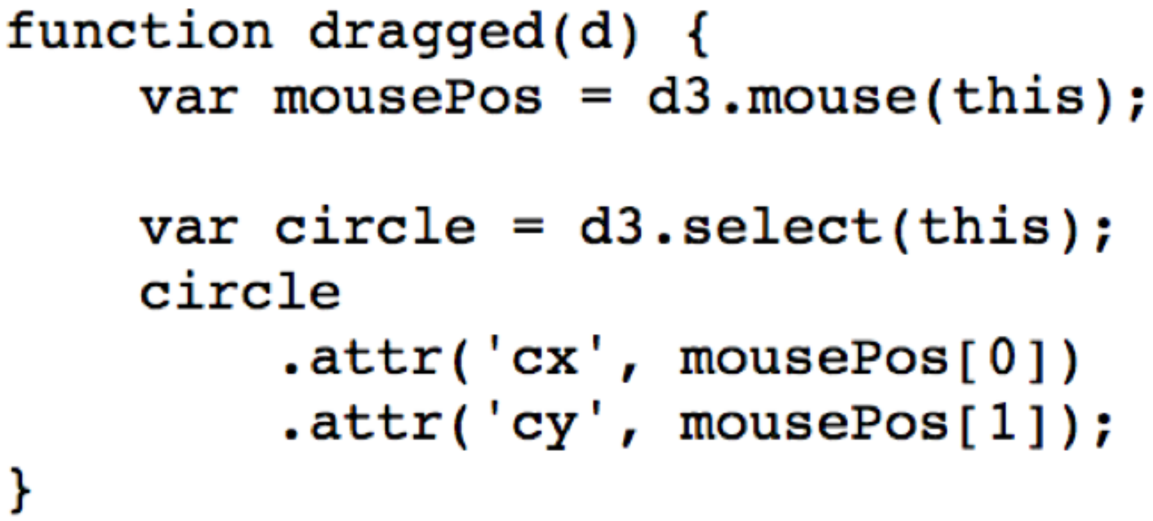
- Create 10 random points

- Create a drag behavior and register a listener *dragged*

- Attach the *drag* behavior to circles

Drag Behavior – d3.drag()

- When we drag a circle, we change the coordinates of the circle according to the position of our mouse



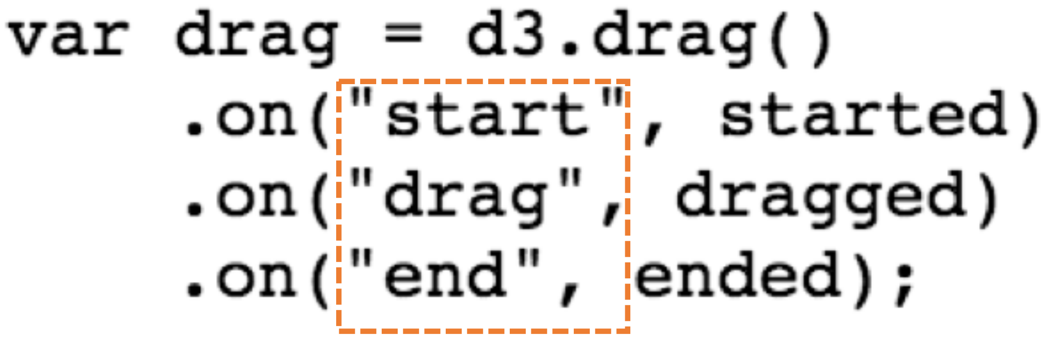
```
function dragged(d) {
    var mousePos = d3.mouse(this);

    var circle = d3.select(this);
    circle
        .attr('cx', mousePos[0])
        .attr('cy', mousePos[1]);
}
```

- Get the mouse position

- Change the coordinates of the circle

Drag Behavior
d3.drag().on(*EventType, listener*)

- d3.drag.on(*EventType, listener*)
- Three types of events
  - start
    - Be triggered at the beginning of the drag behavior
  - drag
    - When the element moves
  - end
    - After the drag behavior ends



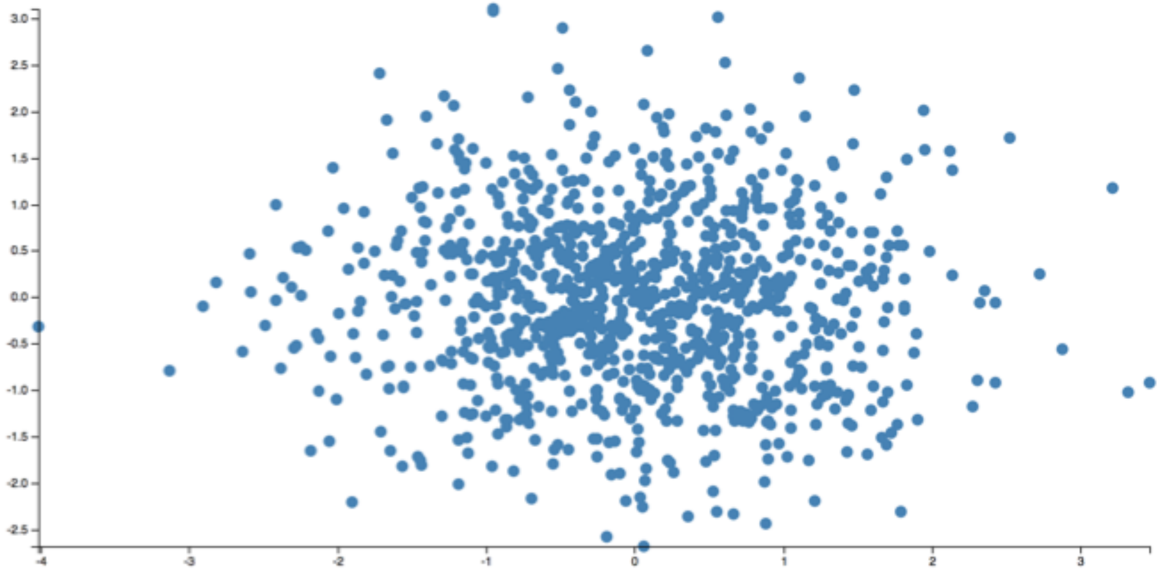
```
var drag = d3.drag()
    .on("start", started)
    .on("drag", dragged)
    .on("end", ended);
```

Zooming and Panning

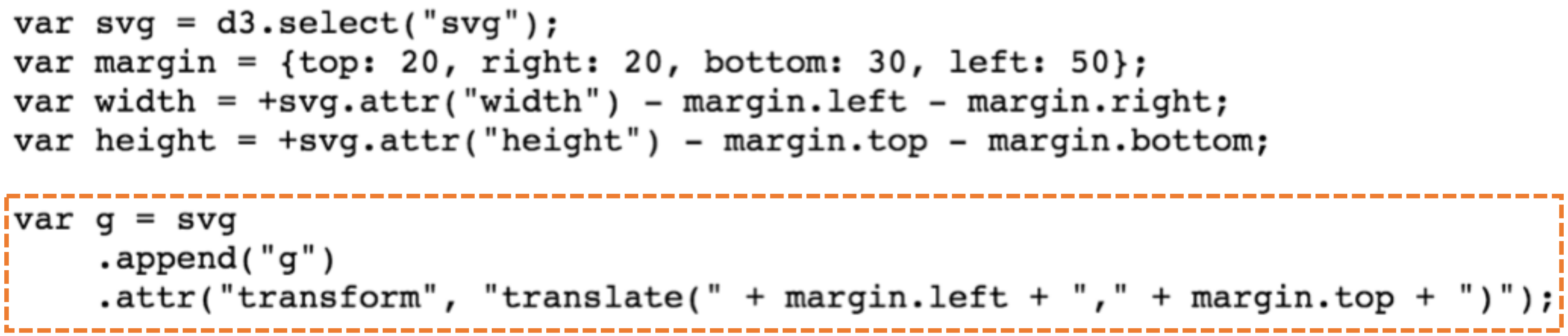
- Zooming and panning and are popular interaction techniques which let the user focus on a region of interest by restricting the view
- Zooming and panning are widely used in web-based mapping, but can also be used with visualizations such as time-series and scatterplots

Zooming and Panning - Scatterplot

- We start from creating a scatterplot which supports zooming and panning
    - Easy to make mistakes
    - We will go through codes in detail



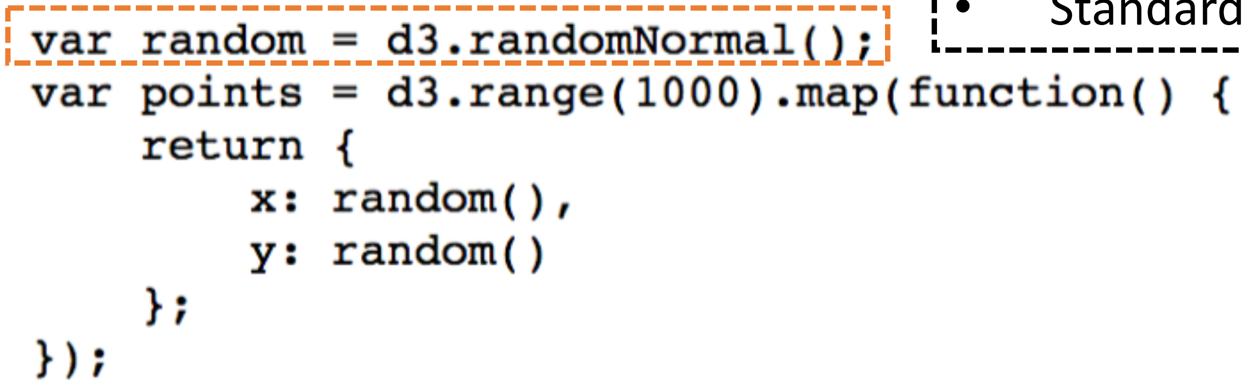
Zooming and Panning - Scatterplot



```
var svg = d3.select("svg");
var margin = {top: 20, right: 20, bottom: 30, left: 50};
var width = +svg.attr("width") - margin.left - margin.right;
var height = +svg.attr("height") - margin.top - margin.bottom;

var g = svg
    .append("g")
    .attr("transform", "translate(" + margin.left + "," + margin.top + ")");
```

- First, we initialize variables
- We draw our scatterplot on a *g* tag

Zooming and Panning - Scatterplot

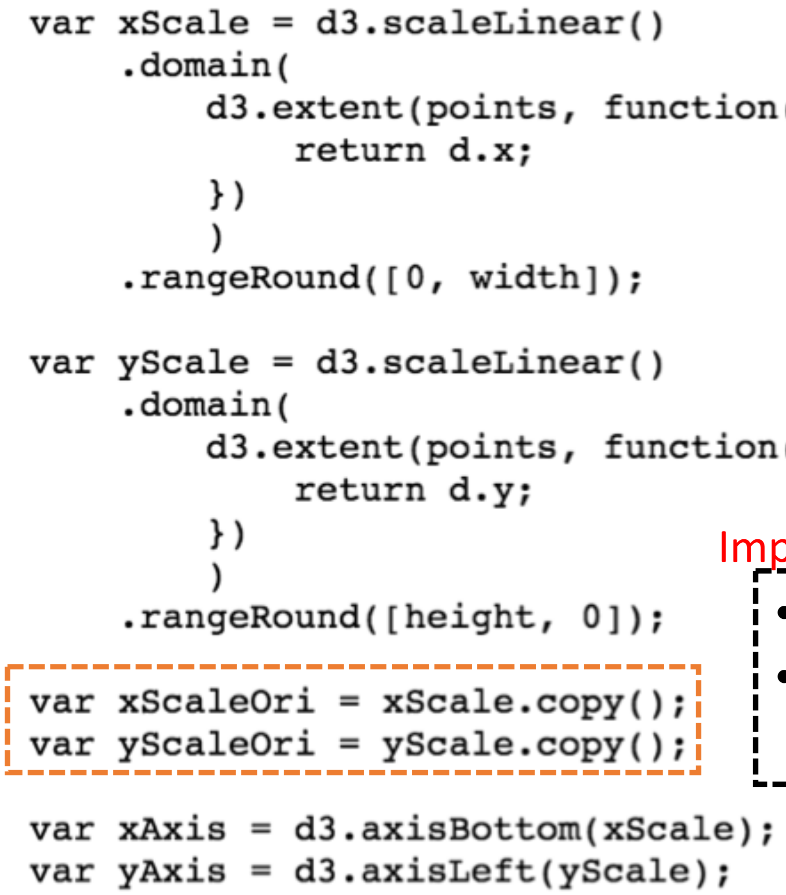


```
var random = d3.randomNormal();
var points = d3.range(1000).map(function() {
    return {
        x: random(),
        y: random()
    };
});
```

- Standard normal distribution

- We generate 1000 points
- Their *x* and *y* coordinates follow the standard normal distribution

Zooming and Panning - Scatterplot



```
var xScale = d3.scaleLinear()
    .domain(
        d3.extent(points, function(d) {
            return d.x;
        })
        )
    .rangeRound([0, width]);

var yScale = d3.scaleLinear()
    .domain(
        d3.extent(points, function(d) {
            return d.y;
        })
        )
    .rangeRound([height, 0]);

var xScaleOri = xScale.copy();
var yScaleOri = yScale.copy();

var xAxis = d3.axisBottom(xScale);
var yAxis = d3.axisLeft(yScale);
```

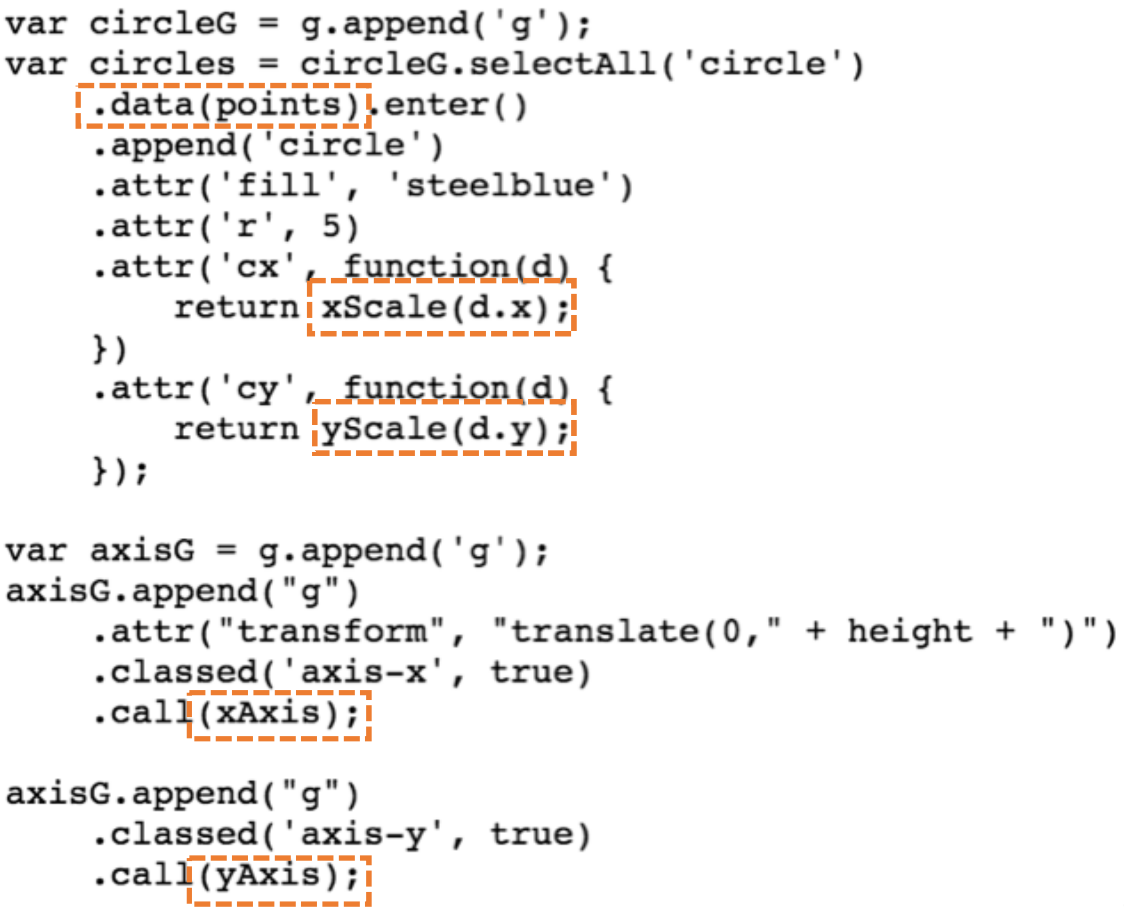
- Create *x* and *y* scales to map coordinates of points on the screen

Important!

- We make a copy of the **ori**ginal scales
- After we zoom and pan, new scales will be created based on these original scales (not transformed scales)

- Create axis generators based on scales

Zooming and Panning - Scatterplot

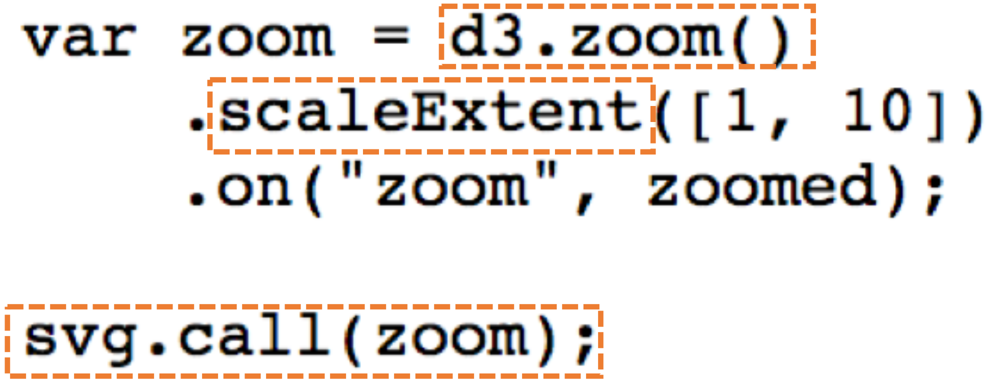


```
var circleG = g.append('g');
var circles = circleG.selectAll('circle')
    .data(points).enter()
    .append('circle')
    .attr('fill', 'steelblue')
    .attr('r', 5)
    .attr('cx', function(d) {
        return xScale(d.x);
    })
    .attr('cy', function(d) {
        return yScale(d.y);
    });

var axisG = g.append('g');
axisG.append("g")
    .attr("transform", "translate(0," + height + ")")
    .classed('axis-x', true)
    .call(xAxis);

axisG.append("g")
    .classed('axis-y', true)
    .call(yAxis);
```

- Draw circles

- Draw axes

Zooming and Panning - Scatterplot

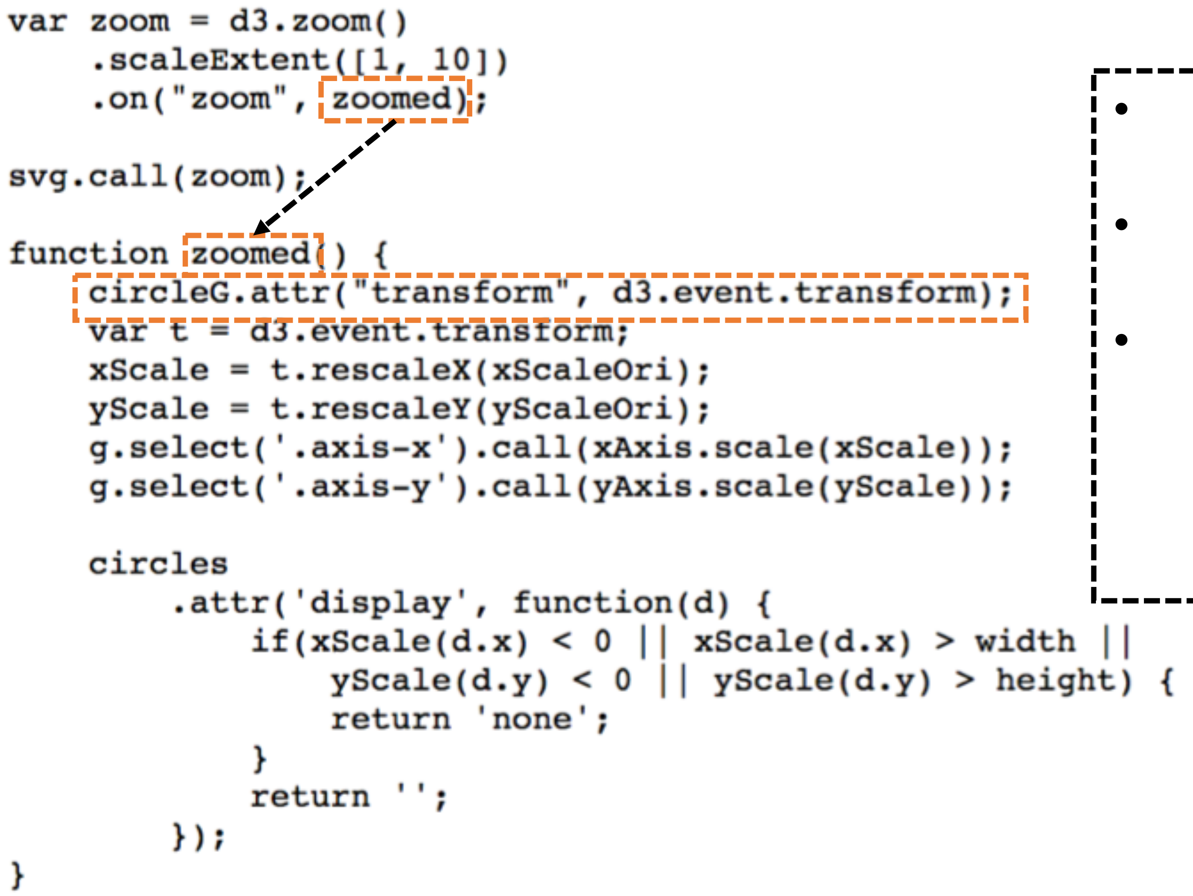


```
var zoom = d3.zoom()
    .scaleExtent([1, 10])
    .on("zoom", zoomed);

svg.call(zoom);
```

- d3.zoom() creates a zooming and panning behavior
- zoom.scaleExtent() sets the min and max zooming scale factors

- Bind zoom behavior with the *svg* element

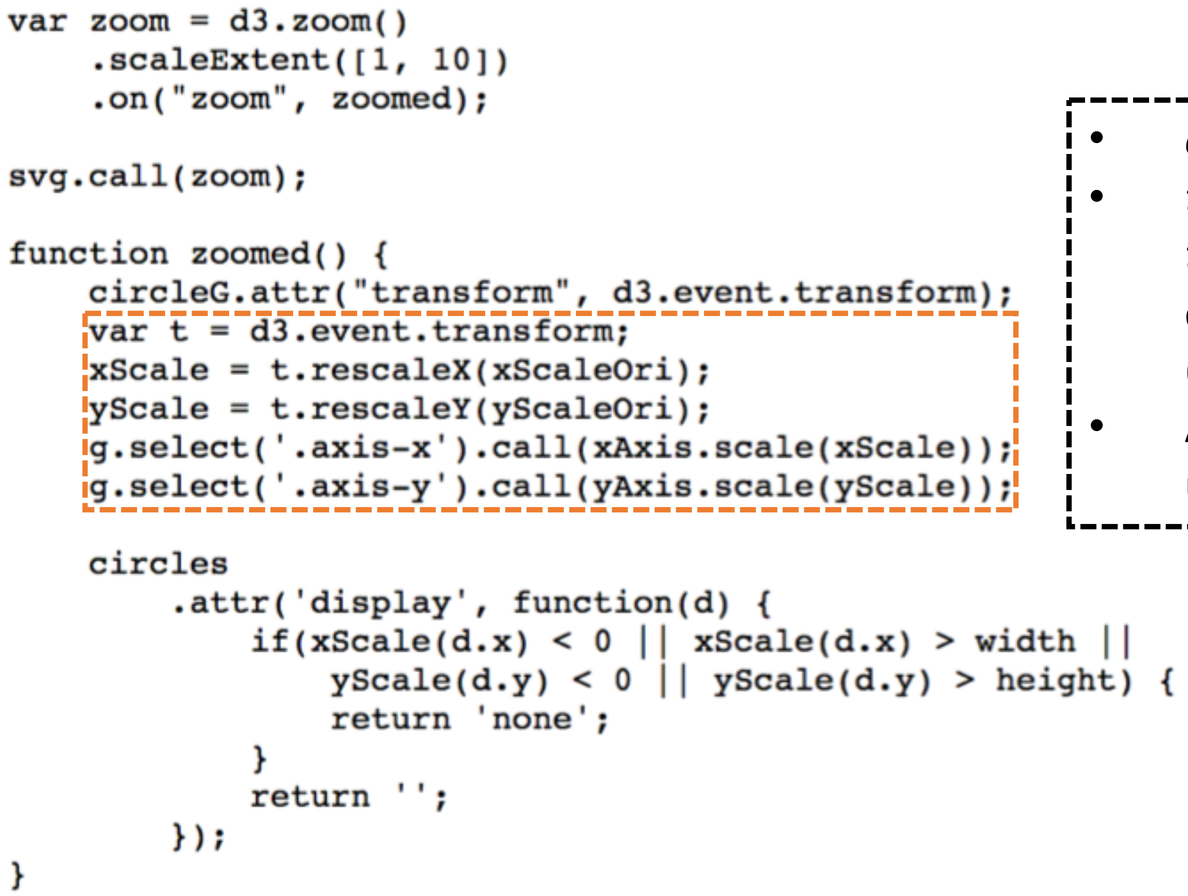
Zooming and Panning - Scatterplot



```
var zoom = d3.zoom()
    .scaleExtent([1, 10])
    .on("zoom", zoomed);

svg.call(zoom);

function zoomed() {
    circleG.attr("transform", d3.event.transform);
    var t = d3.event.transform;
    xScale = t.rescaleX(xScaleOri);
    yScale = t.rescaleY(yScaleOri);
    g.select('.axis-x').call(xAxis.scale(xScale));
    g.select('.axis-y').call(yAxis.scale(yScale));

    circles
        .attr('display', function(d) {
            if(xScale(d.x) < 0 || xScale(d.x) > width ||
                yScale(d.y) < 0 || yScale(d.y) > height) {
                return 'none';
            }
            return '';
        });
}
```

- After we zoom and pan, we have to change the scale (k) and shifting ($\Delta x \ and \ \Delta y$) of points
- *d3.event.transform* can compute $k, \Delta x \ and \ \Delta y$ automatically for you
- When $k = 2, \Delta x = 7 \ and \ \Delta y = 33,$ *d3.event.transform.toString()* outputs “translate(7,33) scale(1)”
  - We can omit *.toString()* when setting *.attr()*

Zooming and Panning - Scatterplot



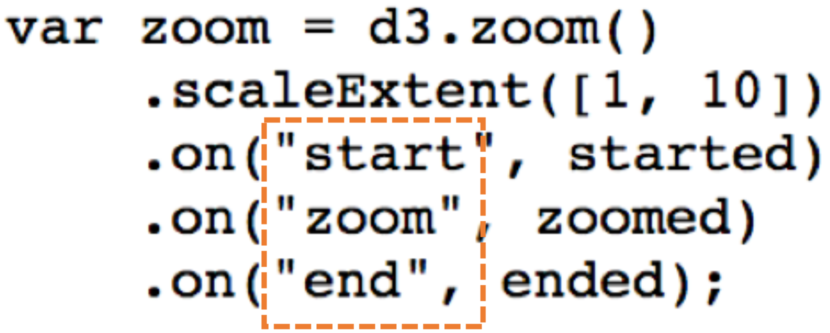
```
var zoom = d3.zoom()
    .scaleExtent([1, 10])
    .on("zoom", zoomed);

svg.call(zoom);

function zoomed() {
    circleG.attr("transform", d3.event.transform);
    var t = d3.event.transform;
    xScale = t.rescaleX(xScaleOri);
    yScale = t.rescaleY(yScaleOri);
    g.select('.axis-x').call(xAxis.scale(xScale));
    g.select('.axis-y').call(yAxis.scale(yScale));

    circles
        .attr('display', function(d) {
            if(xScale(d.x) < 0 || xScale(d.x) > width ||
                yScale(d.y) < 0 || yScale(d.y) > height) {
                return 'none';
            }
            return '';
        });
}
```

- *d3.event.transform* provides several useful functions
- *transform*.rescaleX(x*Scale*) and *transform*.rescaleX(*yScale*) can automatically apply current zooming scale (k) and panning shifting ($\Delta x\ and\ \Delta y$) to the original scales
- After creating new *xScale* and *yScale*, we have to update the axes manually

- We then omit the points which are outside the screen.

Zooming and Panning
d3.zoom() .on(*EventType, listener*)

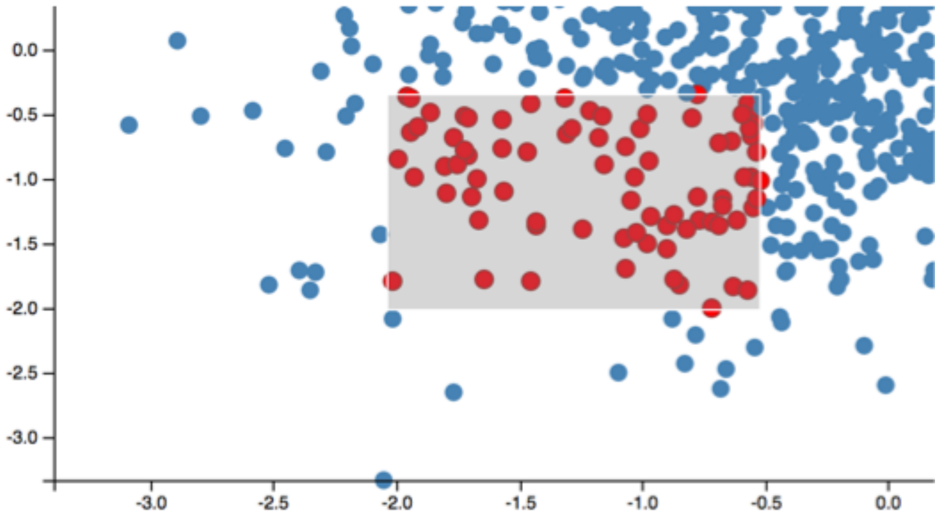
- d3.zoom().on(*EventType, listener*)
- Three types of events
  - start
    - Be triggered at the beginning of the zoom behavior
  - zoom
    - When we zoom the screen
  - end
    - After the zoom behavior ends



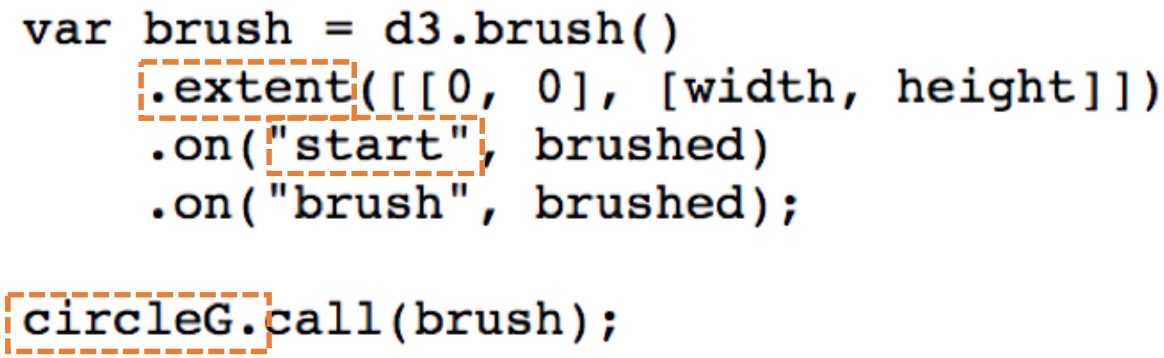
```
var zoom = d3.zoom()
    .scaleExtent([1, 10])
    .on("start", started)
    .on("zoom", zoomed)
    .on("end", ended);
```

Brushing

- Brushing is the interactive specification a one- or two-dimensional selected region using a pointing gesture
  - Such as by clicking and dragging the mouse
- Brushing is often used to select discrete elements, such as dots in a scatterplot or files on a desktop
- We creates a scatterplot which supports brushing



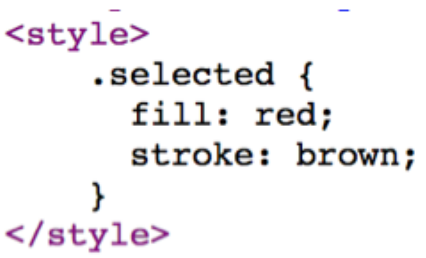
Brushing – d3.brush()



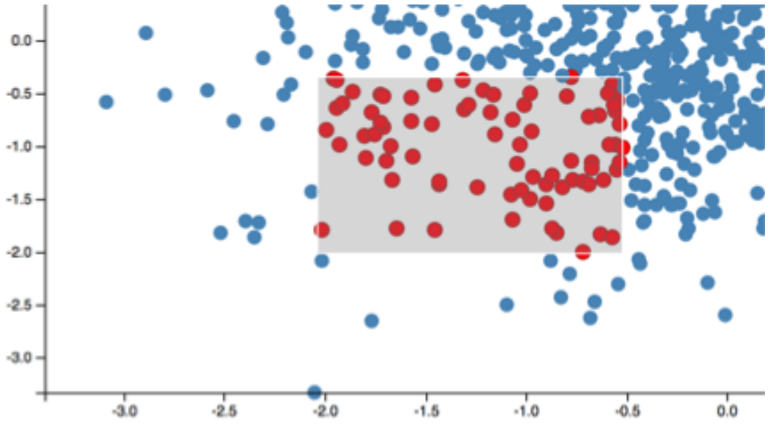
```
var brush = d3.brush()
    .extent([[0, 0], [width, height]])
    .on("start", brushed)
    .on("brush", brushed);

circleG.call(brush);
```

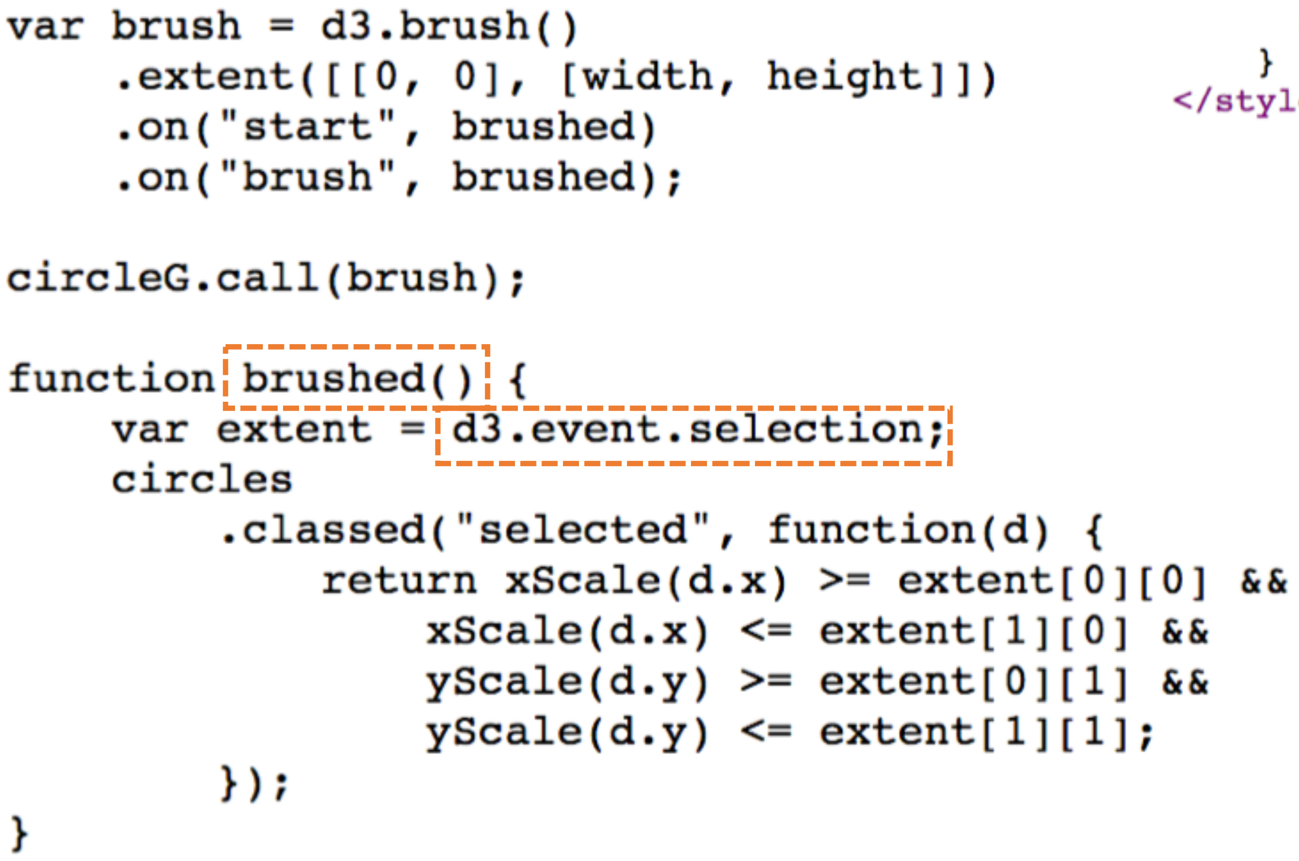
- brush.extent()
  - Sets the brushable area
  - We limit the brushing behavior within the screen
- We have to register a *start* event otherwise the brushing behavior will become weird
  - Just do the same thing as triggering *brush* event
- We then bind brushing behavior with the container of circles because we use brushing to select circles

Brushing – d3.brush()



```
<style>
    .selected {
      fill: red;
      stroke: brown;
    }
</style>
```





```
var brush = d3.brush()
    .extent([[0, 0], [width, height]])
    .on("start", brushed)
    .on("brush", brushed);

circleG.call(brush);

function brushed() {
    var extent = d3.event.selection;
    circles
        .classed("selected", function(d) {
            return xScale(d.x) >= extent[0][0] &&
                xScale(d.x) <= extent[1][0] &&
                yScale(d.y) >= extent[0][1] &&
                yScale(d.y) <= extent[1][1];
        });
}
```

- When brushing, we detect whether circles are within our selection area to determine whether they are selected
- The *d3.event.selection* returns the selection area which is an array [[x0, y0], [x1, y1]], where
    - x0 and x1 are the min and max x-value
    - y0 and y1 are the min and max y-value