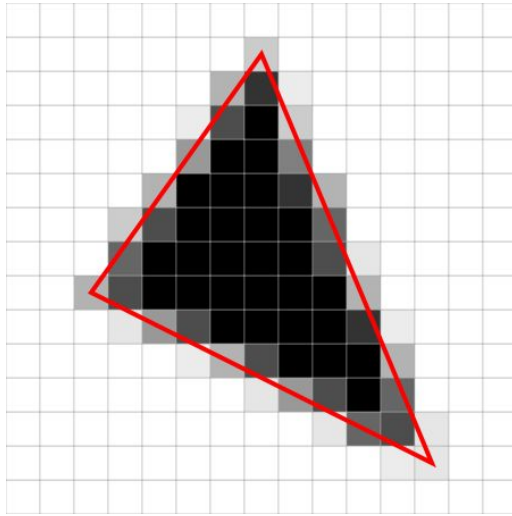


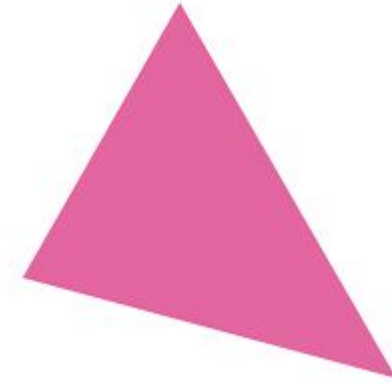
SVG y D3

Introducción a SVG

SVG - Gráficos vectoriales



Rasterizado 20x20px



Vectorial 3 puntos

- Los gráficos rasterizados están codificados en una matriz de píxeles
- Los gráficos vectoriales están codificados en forma de coordenadas y vectores

*SVG es una notación para
representar gráficos
vectoriales*

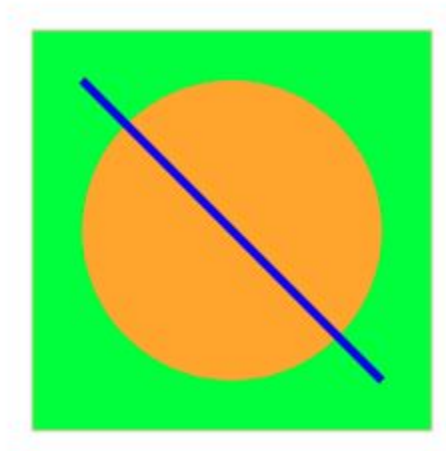
SVG - Scalable Vector Graphics

```
<svg width="391" height="391">  
  <rect x="25" y="25" width="200" height="200" fill="lime" stroke="pink" />  
  <circle cx="125" cy="125" r="75" fill="orange" />  
  <line x1="50" y1="50" x2="200" y2="200" stroke="blue" stroke-width="4" />  
</svg>
```

- Formato basado en sintaxis XML (como HTML)
- Esta estructura nos permite recorrerlos y modificarlos fácilmente mediante código

SVG - Scalable Vector Graphics

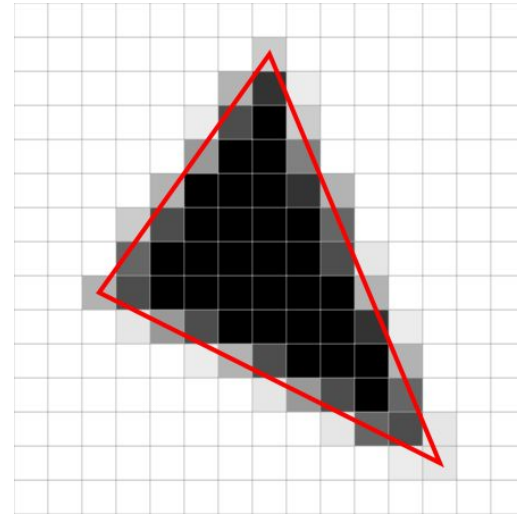
```
<svg width="391" height="391">  
  <rect x="25" y="25" width="200" height="200" fill="lime"/>  
  <circle cx="125" cy="125" r="75" fill="orange" />  
  <line x1="50" y1="50" x2="200" y2="200" stroke="blue" stroke-width="4" />  
</svg>
```



SVG - Gráficos vectoriales vs. gráficos rasterizados



Vectorial



Rasterizado

- Los gráficos rasterizados (matrices de píxeles) se pixelan al escalar
- Los gráficos vectoriales son infinitamente escalables
https://upload.wikimedia.org/wikipedia/commons/d/dd/Infinity_symbol.svg

SVG - Primitivas

Los gráficos SVG se forman a partir de **primitivas** (también llamadas shapes):

```
<svg width="200" height="250">
```

```
<rect x="10" y="10" width="30" height="30"/>
```

```
<circle cx="25" cy="75" r="20"/>
```

```
<ellipse cx="75" cy="75" rx="20" ry="5"/>
```

```
<line x1="10" x2="50" y1="110" y2="150" />
```

```
<polygon points="50 160 55 180 70 180 60 190 65 205 50 195 35 205 40 190 30 180 45 180"/>
```

```
</svg>
```


SVG - Primitiva rect

Nos permite representar rectángulos:

```
<svg width="200" height="250">
```

```
<rect x="10" y="10" width="100" height="100"/>
```

```
</svg>
```



```
<svg width="200" height="250">
```

```
<rect x="10" y="10" width="100" height="200"/>
```

```
</svg>
```

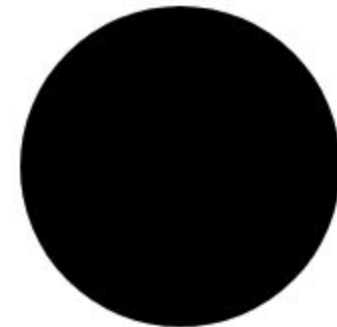


SVG - Primitiva circle

Nos permite representar círculos:

```
<svg width="200" height="250">  
  <circle cx="80" cy="80" r="40"/>  
</svg>
```

```
<svg width="200" height="250">  
  <circle cx="80" cy="80" r="80"/>  
</svg>
```



SVG - Primitiva line

Nos permite representar líneas:

```
<svg width="200" height="250">
```

```
<line x1="10" y1="10" x2="90" y2="90" />
```

```
</svg>
```



```
<svg width="200" height="250">
```

```
<line x1="150" y1="30" x2="10" y2="90" />
```

```
</svg>
```



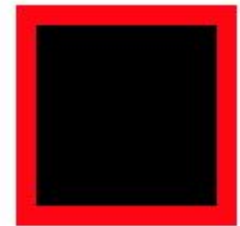
SVG - Dando estilo a una primitiva

Bordes:

```
<svg width="200" height="250">  
  <rect x="10" y="10" width="100" height="100" stroke="red"/>  
</svg>
```



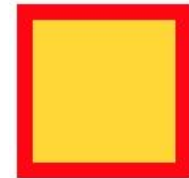
```
<svg width="200" height="250">  
  <rect x="10" y="10" width="100" height="100"  
    stroke="red" stroke-width="10"/>  
</svg>
```



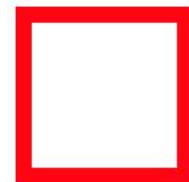
SVG - Dando estilo a una primitiva II

Fill:

```
<svg width="200" height="250">  
  <rect x="10" y="10" width="100" height="100"  
    stroke="red" stroke-width="10" fill="gold"/>  
</svg>
```

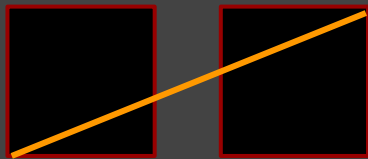


```
<svg width="200" height="250">  
  <rect x="10" y="10" width="100" height="100"  
    stroke="red" stroke-width="10" fill="transparent"/>  
</svg>
```



D3 - Ejercicio primitivas SVG

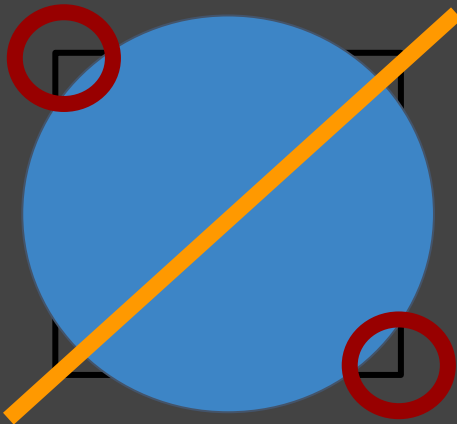
- Ruta ejercicio: /svg/index.html
- Ruta svg: /svg/index.html
- Genera esta forma con svg:



-
- Pista: recuerda que las primitivas tienen que estar dentro de una etiqueta `<svg></svg>`

D3 - Ejercicio primitivas SVG II (extra)

- Ruta ejercicio: </svg/index.html>
- Ruta svg: </svg/index.html>
- Genera la siguiente forma mediante SVG:



*Hay más primitivas, pero de
momento vamos a trabajar
con estas*

D3.js

Qué es d3.js



- Toolkit para manipular documentos basados en datos
- Permite visualizar datos mediante SVG, HTML, CSS, etc.
- La mejor herramienta disponible para data visualization

Qué es d3.js II

- El toolkit incluye
 - Escalas (lineales, logarítmicas, temporales...)
 - Generación de ejes
 - **Sistema de selección de elementos HTML y SVG**
 - Sistema de transiciones
 - **Data-binding**
 - Layouts
 - Simulaciones físicas
 - Proyecciones geográficas
 - Peticiones para cargar ficheros
 - Sistema de eventos propio
 - Gestión de inputs
 -

Qué es d3.js III

- Ejemplos:
 - <https://observablehq.com/@mbostock/tree-of-life>
 - <http://bl.ocks.org/mbostock/1153292>
 - <https://bl.ocks.org/mbostock/1256572>
 - https://www.elconfidencial.com/espana/cataluna/elecciones-catalanas/2017-12-21/elecciones-catalanas-resultados-municipios-provincias-21d_1496088/

Selecciones

D3 - Selecciones

- El primer paso cuando queremos crear una visualización en d3 siempre es realizar una selección.
- Una selección nos permite guardar un elemento HTML/SVG en una variable para poder aplicarle todo tipo de transformaciones.
- Para dibujar una primitiva, necesitaremos seleccionar primero dónde queremos adjuntarla.

D3 - Selecciones II

- Para seleccionar un elemento HTML/SVG...

```
<svg width="200" height="250">  
</svg>
```

-utilizamos el método **d3.select()**

```
let svg = d3.select('svg')
```

D3 - Selecciones III

- Para seleccionar **múltiples** elementos HTML/SVG...

```
<svg width="200" height="250">  
  <rect x="10" y="10" width="100" height="100"/>  
  <rect x="10" y="110" width="100" height="100"/>  
</svg>
```

-utilizamos el método **d3.selectAll()**

```
let rects = d3.selectAll('rect')
```


D3 - Selecciones IV

- Para seleccionar **un** elemento HTML/SVG entre varios...

```
<svg id="svg1" width="200" height="250">  
</svg>
```

```
<svg id="svg2" width="200" height="250">  
</svg>
```

-utilizamos el método **d3.select('#id')**

```
let svg = d3.select('#svg1')  
let svg2 = d3.select('#svg2')
```

D3 - Selecciones V

- Para seleccionar **varios** elementos HTML/SVG pero no todos...

```
<svg width="200" height="250">  
  <rect class="loquesea" x="10" y="10" width="100" height="100"/>  
  <rect x="10" y="110" width="100" height="100"/>  
  <rect class="loquesea" x="10" y="10" width="100" height="100"/>  
  <rect x="10" y="110" width="100" height="100"/>  
</svg>
```

-utilizamos el método **d3.selectAll('.class')**

```
let rects = d3.selectAll('.loquesea')
```

D3 - Selecciones V

- Las selecciones también se pueden aplicar a elementos HTML:

```
<div id="parent">  
  <div class="child"/>  
  <div class="child"/>  
  <div class="child"/>  
</div>
```

```
let parent = d3.select('#parent')  
let children = d3.selectAll('.child')
```

D3 - Ejercicio selección

- Ruta ejercicio: /selections/index.html
- Ruta html: /selections/index.html
- Ruta js: /selections/js/main.js
- Selecciona los elementos de color rojo
- aplica el método **.remove()** a la selecciones.
`selection.remove()`
- Solo deben quedar elementos verdes en pantalla.
- Extra: ¿Si quisiéramos eliminar todo en bloque?

Añadiendo primitivas

D3 - Append

```
<svg id="canvas" width="800" height="600">  
</svg>
```

- Añadimos primitivas utilizando el método `append()`

```
let svg = d3.select('svg')  
let rect = svg.append('rect')  
let circle = svg.append('circle')
```

D3 - Append II

Resultado:

```
<svg id="canvas" width="800" height="600">  
  <rect/>  
  <circle/>  
</svg>
```

- Problema: las primitivas no tienen **atributos** ni **estilo** y, por tanto, no veríamos nada en pantalla
 - **Atributos:** x, y, width, height...
 - **Estilo:** fill, stroke, stroke-width...

D3 - Añadiendo atributos

```
<svg id="canvas" width="800" height="600">  
</svg>
```

```
let svg = d3.select('svg')  
let rect = svg.append('rect')
```

```
rect.attr('class', 'cuadrado')  
rect.attr('x', 10)  
rect.attr('y', 10)  
rect.attr('width', 100)  
rect.attr('height', 100)
```

Resultado:

```
<svg id="canvas" width="800" height="600">  
  <rect class="cuadrado" x="10" y="10" width="100" height="100"/>  
</svg>
```


D3 - Añadiendo estilo

```
<svg id="canvas" width="800" height="600">  
</svg>
```

```
let svg = d3.select(svg)  
let rect = svg.append('rect')
```

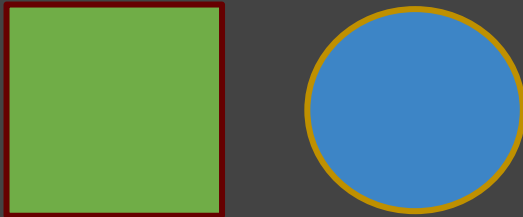
```
rect.style('stroke', 'red')  
rect.style('stroke-width', 3)  
rect.style('fill', 'transparent')
```

Resultado:

```
<svg id="canvas" width="800" height="600">  
  <rect stroke="red" stroke-width="3" fill="transparent"/>  
</svg>
```

D3 - Ejercicio append

- Ruta ejercicio: /append/index.html
- Ruta html: /append/index.html
- Ruta js: /append/js/main.js
- Genera este svg utilizando append:



Añadiendo el elemento svg

D3 - Añadiendo el elemento svg

Por lo general, el elemento svg no nos lo encontraremos puesto dentro del html, tendremos que añadirlo nosotros:

```
<div id="parent" style="width: 800px; height: 600px;">  
</div>
```

```
let container = d3.select('#parent')  
let svg = container.append(svg)  
svg.attr('id', 'canvas')  
svg.attr('width', 800)  
svg.attr('height', 600)
```

Resultado:

```
<div id="parent" style="width: 800px; height: 600px;">  
  <svg id="canvas" width="800" height="600">  
  </svg>  
</div>
```

*Problema: estamos
hardcodeando el width y el
height*

D3 - Capturando ancho y alto de un elemento HTML

Podemos acceder al width y height de un elemento HTML:

```
<div id="parent" style="width: 800px; height: 600px;">  
</div>
```

```
let container = d3.select( '#parent' )  
let containerWidth = container.node().offsetWidth  
let containerHeight = container.node().offsetHeight
```

- **Container.node()** nos devuelve una **selección HTML**
- Las selecciones HTML nos permiten acceder a atributos de elementos HTML
- No entraremos en las selecciones HTML pero en este caso es útil poder tener acceso a las dimensiones

D3 - Añadiendo el elemento svg II

Podemos acceder al width y height de un elemento HTML:

```
<div id="parent" style="width: 800px; height: 600px;">  
</div>
```

```
let container = d3.select('#parent')
```

```
let containerWidth = container.node().offsetWidth
```

```
let containerHeight = container.node().offsetHeight
```

```
let svg = container.append(svg)
```

```
svg.attr('id', 'canvas')
```

```
svg.attr('width', containerWidth)
```

```
svg.attr('height', containerHeight)
```

Resultado:

```
<div id="parent" style="width: 800px; height: 600px;">  
  <svg id="canvas" width="800" height="600">  
  </svg>  
</div>
```

D3 - Ejercicio cuadrados

- Ruta ejercicio: /cuadrados/index.html
- Ruta html: /cuadrados/index.html
- Ruta js: /cuadrados/js/main.js
- Selecciona el div contenedor que hay en index.html
- Añade un elemento svg que tenga las mismas dimensiones que el contenedor HTML

D3 - Ejercicio cuadrados II

- Ruta ejercicio: /cuadrados/index.html
- Ruta html: /cuadrados/index.html
- Ruta js: /cuadrados/js/main.js
- Partiendo del ejercicio anterior:
 - Añade 4 cuadrados, uno debajo de otro
 - Dimensiones cuadrado: 30x20
 - Margen entre cuadrados: 20
 - Margen izquierda: 100
 - Color cuadrados: #B8EA10

El proceso es muy pesado

D3 - Mejorando el workflow

- Soluciones:
 - Modularizar utilizando funciones
 - Method chaining
 - Data-binding

D3 - Ejercicio cuadrados III

- Ruta ejercicio: /cuadrados/index.html
- Ruta html: /cuadrados/index.html
- Ruta js: /cuadrados/js/main.js
- Partiendo del ejercicio anterior:
 - Separa el código en tres bloques:
 - Una función **createSVG()** que recibe el selector (“#container”) y le añade el elemento svg con las mismas dimensiones
 - Una función **build()** que añade los cuadrados
 - Una función **main()** que llama a createSVG y a build
- Extra: Variables globales

Method chaining

D3 - Method chaining

- Consiste en que las funciones que aplican una transformación a un elemento, **devuelvan siempre ese elemento en el return**
- De esa forma, podemos encadenar las transformaciones:

```
ball.moveTo(10, 10).scale(5).moveTo(0, 0)
```

- Equivale a:

```
ball.moveTo(10, 10)  
ball.scale(5)  
ball.moveTo(0, 0)
```

D3 - Method chaining II

```
let rect = svg.append('rect')
rect.attr('x', 100)
rect.attr('y', 20)
rect.attr('width', 30)
rect.attr('height', 20)
rect.style('fill', '#B8EA10')
```

- Equivalente method chaining

```
let rect = svg.append('rect')

rect
  .attr('x', 100)
  .attr('y', 20)
  .attr('width', 30)
  .attr('height', 20)
  .style('fill', '#B8EA10')
```

D3 - Method chaining III

```
let rect = svg.append('rect')
```

rect

```
.attr('x', 100)
```

```
.attr('y', 20)
```

```
.attr('width', 30)
```

```
.attr('height', 20)
```

```
.style('fill', '#B8EA10')
```

- Equivalente:

```
let rect = svg.append('rect')
```

```
.attr('x', 100)
```

```
.attr('y', 20)
```

```
.attr('width', 30)
```

```
.attr('height', 20)
```

```
.style('fill', '#B8EA10')
```


D3 - Ejercicio cuadrados IV

- Ruta ejercicio: /cuadrados/index.html
- Ruta html: /cuadrados/index.html
- Ruta js: /cuadrados/js/main.js
- Aplicar method chaining a los cuadrados

Insertar elementos uno a uno está muy bien pero...

Data binding

D3 - Data binding

- Data binding nos permite crear una relación entre nuestros datos y los elementos SVG

```
let datos = ['uno', 'dos', 'tres']
```

```
let rects = svg.selectAll('rect')  
  .data(datos)
```

- Hemos asignado **3 datos** a la selección

*¿Que nos permite este
vínculo?*

D3 - Enter / exit

- Dos operaciones:
 - **Enter:** para añadir elementos SVG que falten
 - **Exit:** para eliminar elementos SVG que sobren

D3 - Enter

- Nos permite **añadir** elementos SVG a datos que todavía no tienen un elemento asignado

```
let datos = ['uno', 'dos', 'tres']  
  
let rects = svg.selectAll('rect') // Selección vacía, no hay rects  
    .data(datos)
```

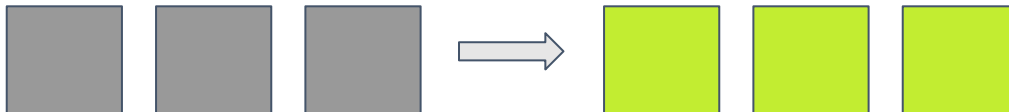
- Ejemplo de uso:

```
let rects = svg.selectAll('rect')  
    .data(datos) // asignamos datos  
    .enter() // entramos en modo enter  
    .append('rect') // Añadimos un rectángulo  
        .attr('x', 10)  
        .attr('y', 10)
```

D3 - Enter II

```
let datos = ['uno', 'dos', 'tres']  
  
let rects = svg.selectAll('rect')  
  .data(datos) // asignamos datos  
  .enter() // entramos en modo enter  
  .append('rect') // Añadimos un rectángulo  
    .attr('x', 10)  
    .attr('y', 10)
```

- Esta operación va a añadir **3 rects** (todos en la misma posición, ya lo arreglaremos)



D3 - Enter III

```
let datos = ['uno', 'dos', 'tres']  
  
let rects = svg.selectAll('rect')  
  .data(datos) // asignamos datos  
  .enter() // entramos en modo enter  
  .append('rect') // Añadimos un rectángulo  
    .attr('x', 10)  
    .attr('y', 10)
```

- Qué pasaría si la selección no estuviera vacía? Es decir **hay 1 rect** en el svg



D3 - Enter IV

```
let datos = ['uno', 'dos', 'tres']  
  
let rects = svg.selectAll('rect')  
  .data(datos) // asignamos datos  
  .enter() // entramos en modo enter  
  .append('rect') // Añadimos un rectángulo  
    .attr('x', 10)  
    .attr('y', 10)
```

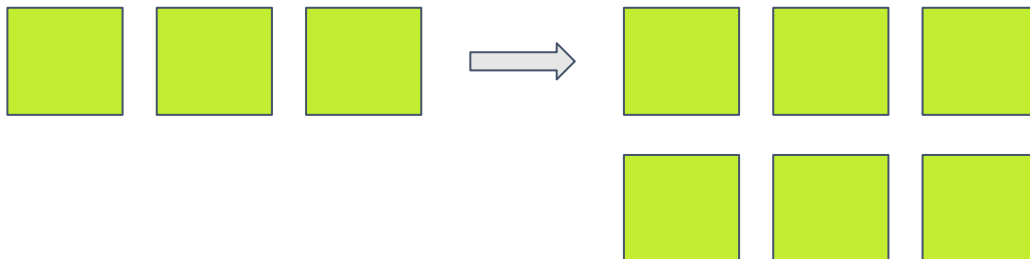
- Qué pasaría si ya hay **3 rects** en el svg?
- Nada



D3 - Enter II

```
let datos = ['uno', 'dos', 'tres']  
  
let rects = svg.selectAll('circle')  
  .data(datos) // asignamos datos  
  .enter() // entramos en modo enter  
  .append('rect') // Añadimos un rectángulo  
    .attr('x', 10)  
    .attr('y', 10)
```

- Qué pasa ahora?



D3 - Enter II

```
let datos = ['uno', 'dos', 'tres']  
  
let rects = svg.selectAll('circle')  
  .data(datos) // asignamos datos  
  .enter() // entramos en modo enter  
  .append('rect') // Añadimos un rectángulo  
    .attr('x', 10)  
    .attr('y', 10)
```

- Qué pasa ahora?



D3 - Exit

- Nos permite **seleccionar** los elementos SVG que no tienen datos asignados, **generalmente para deshacernos de ellos**

```
let datos = [ 'uno' ]
```

```
let rects = svg.selectAll('rect') // Hay 3 rects  
  .data(datos)
```

- Aplicamos exit:

```
let rects = svg.selectAll('rect')  
  .data(datos) // asignamos datos  
  .exit() // entramos en modo exit  
  .remove() // Aplicamos remove a cada elemento
```

D3 - Enter II

```
let datos = ['uno']  
  
let rects = svg.selectAll('rect')  
  .data(datos) // asignamos datos  
  .exit() // entramos en modo exit  
  .remove() // Aplicamos remove a cada elemento
```

- Se eliminan 2 rectángulos

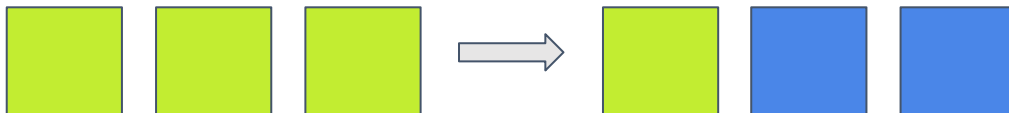


D3 - Enter II

- Exit solo selecciona



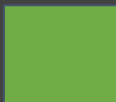

```
let datos = ['uno']  
  
let rects = svg.selectAll('rect')  
  .data(datos) // asignamos datos  
  .exit() // entramos en modo exit  
  .style('fill', 'blue') // Aplicamos style a cada elemento
```

- Podemos hacer con esa selección lo que queramos







D3 - Ejercicio data-binding

Cuántos cuadrados añadimos con **enter** y **append**:

- `[1, 1, 1]` ,  x 0
- `[1, 1, 2, 4]` ,  x 2
- `[[1, 2], 1, [3]]` ,  x 0
- `[{name: "alex", age: "30"}, [2, 2, [2, [2]]], {}, []]` ,  x 1

D3 - Ejercicio data-binding II

Cuántos cuadrados eliminamos con **exit** y **remove**:

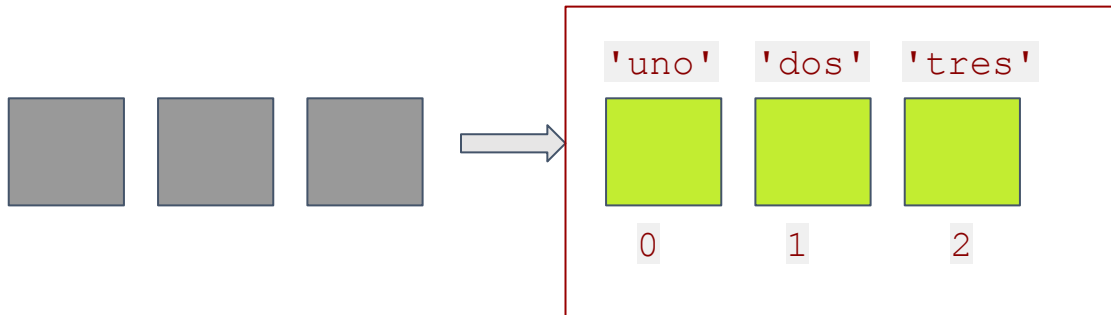
- `["dos", 2, 1]`,  x 5
- `[1, 1, 2, 4]`,  x 4
- `[[1, 1], {}, [3]]`,  x 3
- `[{name: "alex", age: "30"}, [2, 2, [2, [2]]], {}, []]`,  x 1

La ventaja del data-binding

D3 - Data-binding

- **.data()** asigna cada dato a cada elemento
- También les asigna un índice numérico

```
let datos = ['uno', 'dos', 'tres']  
  
let rects = svg.selectAll('rect')  
  .data(datos) // asignamos datos  
  .enter() // entramos en modo enter  
  .append('rect') // Añadimos un rectángulo  
    .attr('x', 10)  
    .attr('y', 10)
```

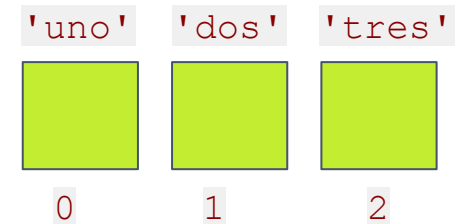


*Cómo accedemos a los datos
asignados a cada elemento?*

D3 - Data-binding II

- attr y style también permiten asignar funciones que reciben los datos asignados

```
let datos = ['uno', 'dos', 'tres']  
  
let rects = svg.selectAll('rect')  
  .data(datos) // asignamos datos  
  .enter() // entramos en modo enter  
  .append('rect') // Añadimos un rectángulo  
    .attr('x', function(d, i){return i * 10})  
    .attr('y', 10)
```



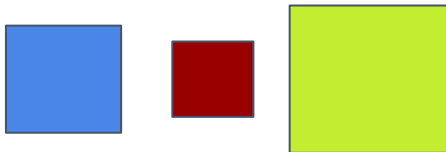
- **d** es el **dato**
- **i** es el **índice**
- **return** devuelve el valor que asignamos a **x**

D3 - Data-binding II

- Ejemplo de uso

```
let datos = [  
  {size: 20, color: "blue" },  
  {size: 10, color: "red" },  
  {size: 30, color: "green" }  
]
```

```
let rects = svg.selectAll('rect')  
  .data(datos) // asignamos datos  
  .enter() // entramos en modo enter  
  .append('rect') // Añadimos un rectángulo  
    .attr('width', function(d, i){return d['size']})  
    .attr('height', function(d, i){return d.size})  
    .style('fill', function(d, i){return d.color})
```



D3 - Data-binding II

- Ejemplo de uso

```
let datos = [  
  {size: 20, color: "blue" },  
  {size: 10, color: "red" },  
  {size: 30, color: "green" }  
]
```

```
let rects = svg.selectAll('rect')  
  .data(datos) // asignamos datos  
  .enter() // entramos en modo enter  
  .append('rect') // Añadimos un rectángulo  
    .attr('x', 20)  
    .attr('y', function(d, i) {return i * 30} )
```



D3 - Ejercicio cuadrados V

- Ruta ejercicio: /cuadrados/index.html
- Ruta html: /cuadrados/index.html
- Ruta js: /cuadrados/js/main.js
- Partiendo de la anterior versión:
 - Crea una array que contenga cuatro colores
 - Crea los cuadrados, pero esta vez utilizando data-binding
 - Cada cuadrado debe tener el color correspondiente al dato
 - Pista: utiliza el índice (i) para posicionar los cuadrados

*Ya tenemos una base de
código más elegante*

Primitiva texto

D3 - Primitiva texto

- Igual que el resto de primitivas
- nuevo atributo **.text()** que contiene el texto

```
.append('text')  
  .attr('class', 'text')  
  .attr('x')  
  .attr('y')  
  .style('font-weight', 500) // grosor  
  .style('font-family', 'Arial') // familia  
  .style('fill', 'red') // color  
  .text('Contenido del texto') // No se hace a través de attr
```

D3 - Primitiva texto

- **.text()** funciona con **data-binding**, igual que attr y style

```
.append('text')  
  .attr('class', 'text')  
  .attr('x')  
  .attr('y')  
  .style('font-weight', 500) // grosor  
  .style('font-family', 'Arial') // familia  
  .style('fill', 'red') // color  
  .text(function(d,i){  
    return d.name  
  })
```

cargar un json con d3

D3 - Loading JSON data

../data/rects.json

```
[  
  {"label": "Elemento A", "value": 2, "color": "teal"},  
  {"label": "Elemento B", "value": 3, "color": "orange"},  
  {"label": "Elemento C", "value": 7, "color": "purple"},  
  {"label": "Elemento D", "value": 4, "color": "red"},  
]
```

Cómo cargar un json con d3 (callback):

```
d3.json('../data/rects.json').then(function(data) {  
  console.log(data)  
})
```

D3 - Loading JSON data II

Cuidado con la asincronía:

```
console.log('before')
d3.json('../data/rects.json').then(function(data) {
  console.log('middle')
})
console.log('after')
```

```
// before
// after
// middle
```

D3 - Loading JSON data III

Solución:

```
console.log('before')
d3.json('../data/rects.json').then(function(data) {
  console.log('middle')
  console.log('after')
})
```

```
// before
// middle
// after
```

Todo lo que dependa de la recepción de los datos debe estar dentro del callback de **d3.json()**

D3 - Loading JSON data III

Podemos seguir modularizando el código:

```
d3.json('../data/rects.json').then(drawData)
```

```
function drawData(data) {  
  console.log('middle')  
  finish()  
}
```

```
function finish() {  
  console.log('after')  
}
```

D3 - Diagrama de barras (ejercicio largo)

- Ruta datos: /cuadrados/data/pollution.json
- Partiendo de la anterior versión:
 - Carga el fichero de datos con d3.json()
 - Genera tantos rectángulos como datos
 - Rectángulos de color #B8EA10
 - Genera textos a la izquierda de los rectángulos con los nombres de las ciudades
 - El ancho de los cuadrados tiene que depender de el nivel de polución. 1 equivale a 50 px, 2 equivale a 100px, etc.

Animando con d3

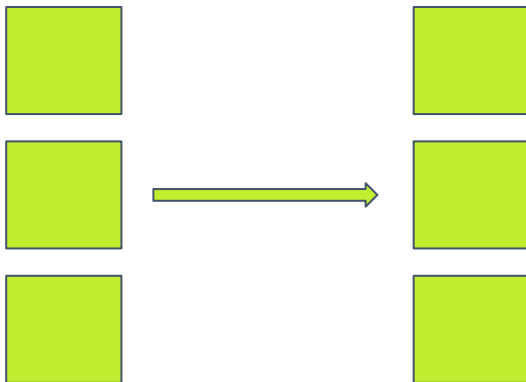
D3 - Transitions

Queremos cambiar de este estado...

```
let rects = svg.selectAll('rect') // 3 datos
  .data(datos)
  .enter()
  .append('rect') // Añadimos un rectángulo
  .attr('x', 20)
```

... a este estado:

```
rects
  .attr('x', 80)
```



Transition

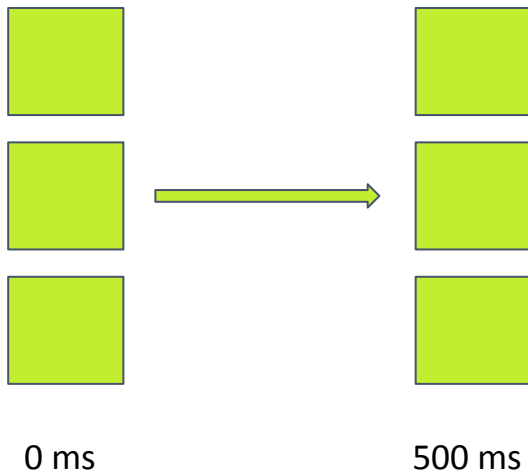
D3 - Transitions II

Queremos cambiar de este estado...

```
let rects = svg.selectAll('rect') // 3 datos
  .data(datos)
  .enter()
  .append('rect') // Añadimos un rectángulo
  .attr('x', 20)
```

... a este estado:

```
rects
  .transition().duration(500) // milisegundos
  .attr('x', 80)
```

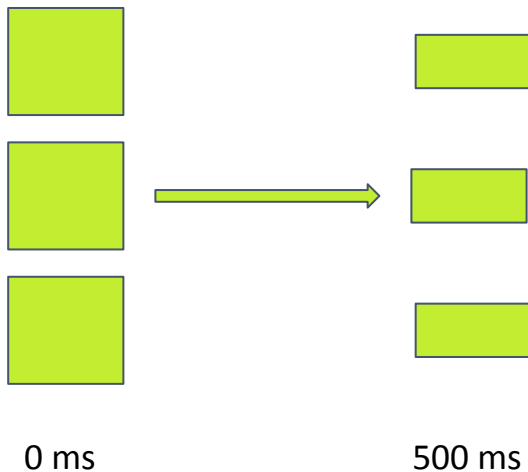


D3 - Transitions III

Podemos cambiar múltiples atributos:

```
rects  
  .attr('x', 80)  
  .attr('height', 50)
```

```
rects  
  .transition().duration(500) // milisegundos  
  .attr('x', 80)  
  .attr('height', 25)
```

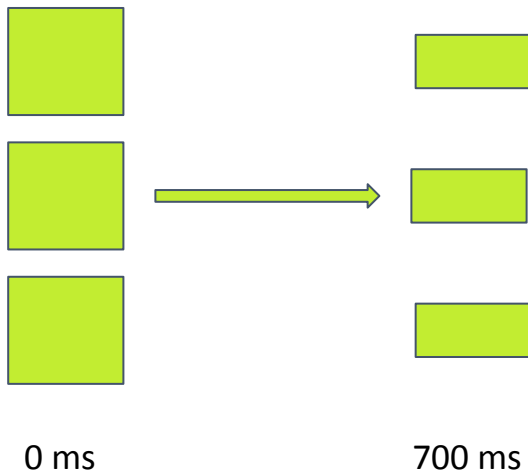


D3 - Transitions IV

Podemos establecer cuánto tarda la animación en empezar con **delay()**

```
rects
  .attr('x', 80)
  .attr('height', 50)
```

```
rects
  .transition().duration(500).delay(200)
  .attr('x', 80)
  .attr('height', 25)
```

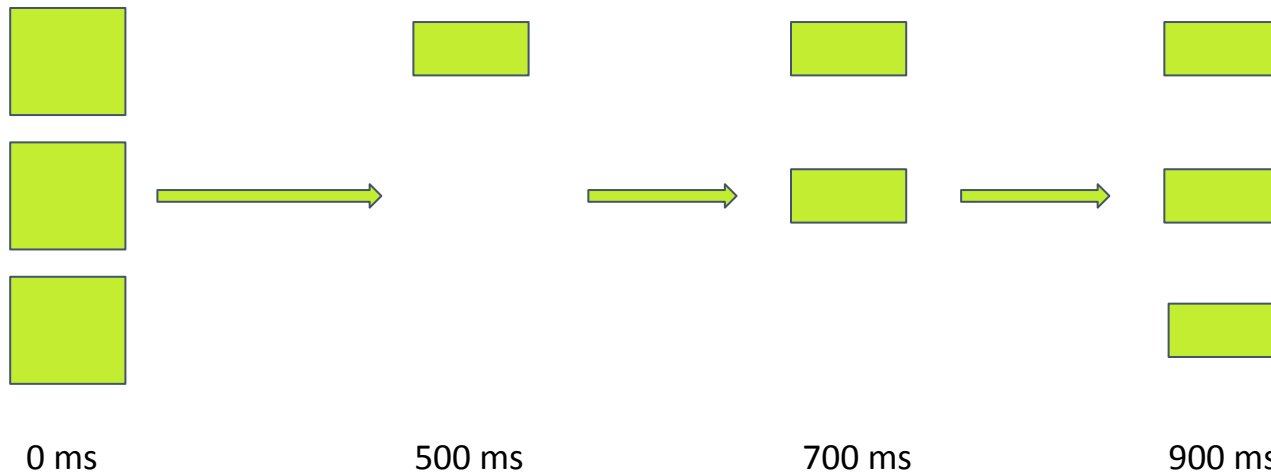


D3 - Transitions V

.duration() y **.delay()** también funcionan con data-binding

```
rects
  .attr('x', 80)
  .attr('height', 50)
```

```
rects
  .transition().duration(500).delay(function(d,i){return i * 200})
  .attr('x', 80)
  .attr('height', 25)
```



D3 - Diagrama de barras II

- Partiendo de la anterior versión:
 - Las barras deben comenzar con ancho cero
 - Las barras deben terminar con el ancho adecuado
 - La duración debe ser de: 0.8 segundos
 - Las barras deben aparecer una detrás de otra con 80 milisegundos de margen
 - `.style('opacity', 0-1)`

D3 - setTimeout

Podemos lanzar eventos después de un delay en Javascript:

```
setTimeout(function() {  
    console.log('hola')  
}, 3000)
```

```
// Imprimirá hola a los 3 segundos
```

D3 - setTimeout

Podemos generar números aleatorios en Javascript usando **Math.random()**:

<code>Math.random()</code>	<code>// 0-1</code>
<code>Math.random() * 10</code>	<code>// 0-10</code>
<code>Math.random() * 10) - 5</code>	<code>// (-5)-5</code>

Podemos utilizar la aleatoriedad para generar datos de prueba

D3 - Diagrama de barras III (extra)

- Ruta datos: /cuadrados/data/pollution.json
- Partiendo del ejercicio anterior:
 - Modifica el json con los niveles de pollution ligeramente alterados (+-2 niveles de polución)
 - A los 3 segundos, actualiza los datos con los contenidos del nuevo json
 - Las barras deben actualizarse con un delay de 80ms entre ellas
 - Pista: `2.2342342.toFixed(1) // 2.2`
- Extra: `setInterval`