VISUALISATIONS COMPLEXES AVEC D3.JS

ATELIER THÉMATIQUE EN VISUALISATION DE DONNÉES

Antoine Béland

11 octobre 2018

PLAN DE L'APRÈS-MIDI

1. Mécanismes de mise à jour dans D3.js
2. Transitions avec D3.js
3. Modèles de visualisation dans D3.js
4. Mise en pratique

D3.JS — MÉCANISMES DE MISE À JOUR

MÉCANISMES DE MISE À JOUR

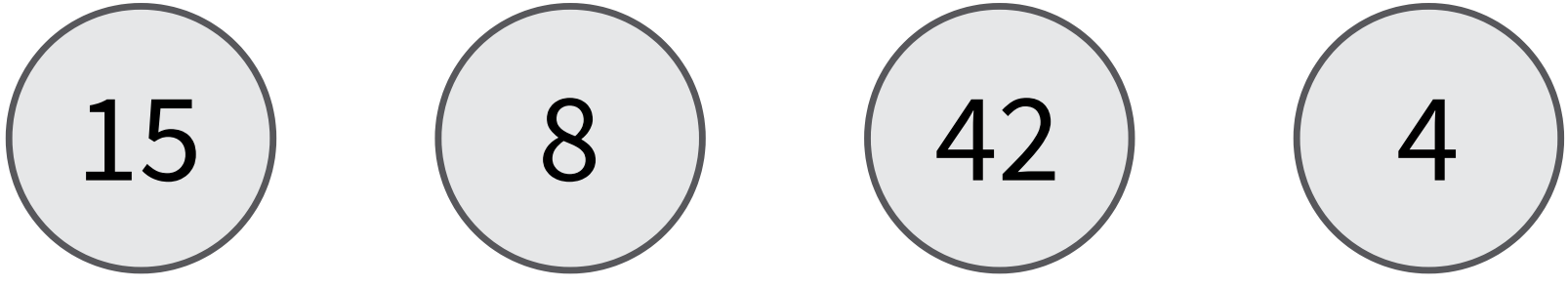
- Il est souvent nécessaire de modifier les données utilisées pendant la présentation d'une visualisation
- D3.js intègre plusieurs mécanismes pour réaliser cette mise à jour de manière simple et efficace
- En fonction des données à utiliser, des éléments devront être **créés**, **supprimés** ou **modifiés**.

MÉCANISMES DE MISE À JOUR

1. **Sélectionner** tous les éléments à mettre à jour sur la visualisation et leur associer les données courantes
2. **Créer** les nouveaux éléments avec la fonction `enter`
3. **Mettre à jour** les nouveaux éléments et ceux existants avec la fonction `merge`
4. **Supprimer** les éléments superflus avec la fonction `exit`

MISE À JOUR — DÉMONSTRATION

- Supposons que l'on souhaite utiliser l'ensemble de données suivant pour créer un *bar chart*:



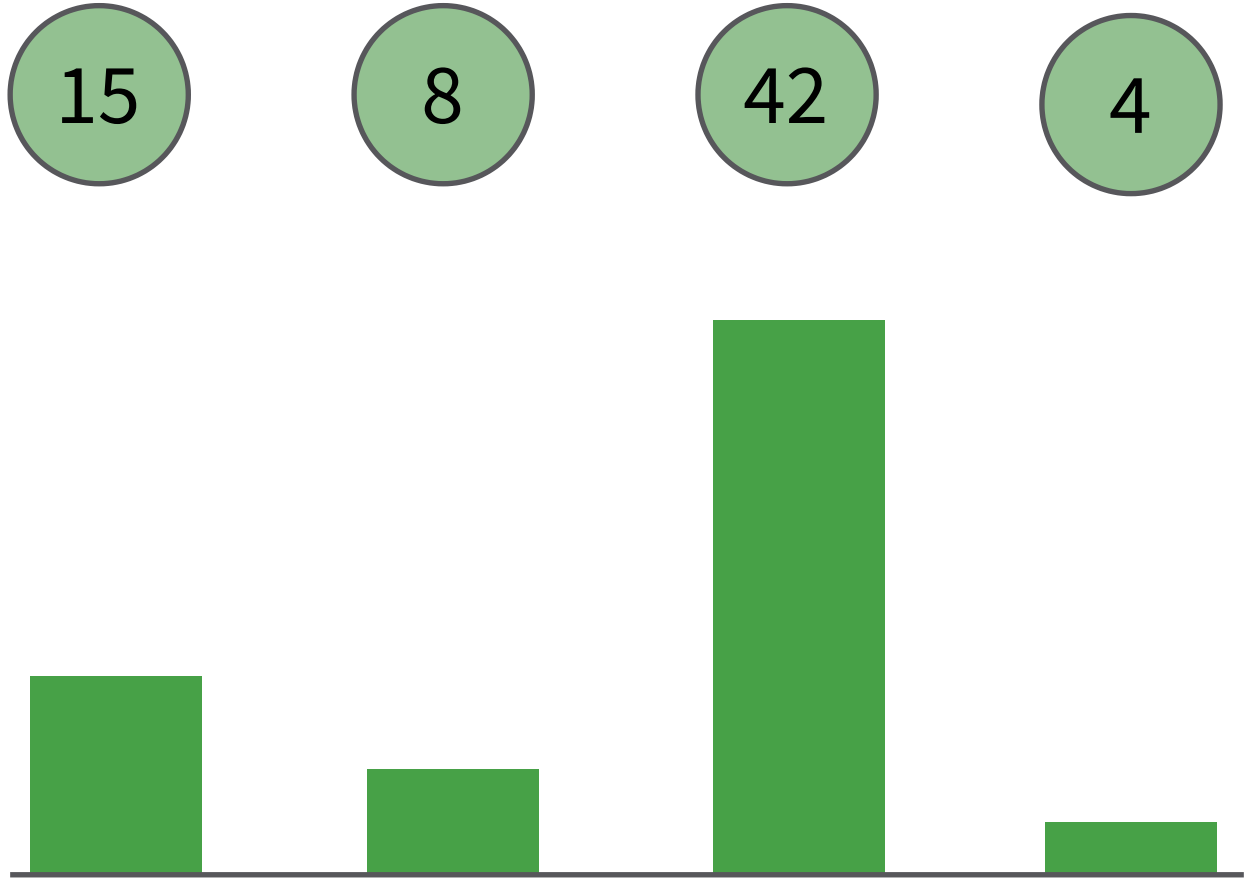
MISE À JOUR — DÉMONSTRATION

- En termes de code, cela pourrait se traduire par les lignes suivantes:

```
const svg = d3.select('svg')
const data = [15, 8, 42, 4];

// (1) Sélection et association des données
const rectangles = svg.selectAll('rect')
  .data(data);
```

MISE À JOUR — DÉMONSTRATION

- Puisqu'il n'y avait pas de barre auparavant, on doit entrer dans la fonction **`enter`** pour les créer



MISE À JOUR — DÉMONSTRATION

- En termes de code, cela se traduit en utilisant la fonction **`enter`** une fois la sélection effectuée

```
// (1) Sélection et association des données
const rectangles = svg.selectAll('rect')
  .data(data);

// (2) Création des nouveaux éléments
const rectanglesCreated = rectangles.enter()
  .append('rect')
  .attr('height', d => d)
  .style('fill', 'green');
```

MISE À JOUR — DÉMONSTRATION

- Supposons que l'on souhaite maintenant utiliser un autre ensemble de données pour créer le *bar chart*

MISE À JOUR — DÉMONSTRATION

- Une nouvelle barre sera créée avec la fonction `enter` tandis que l'autre sera uniquement mise à jour

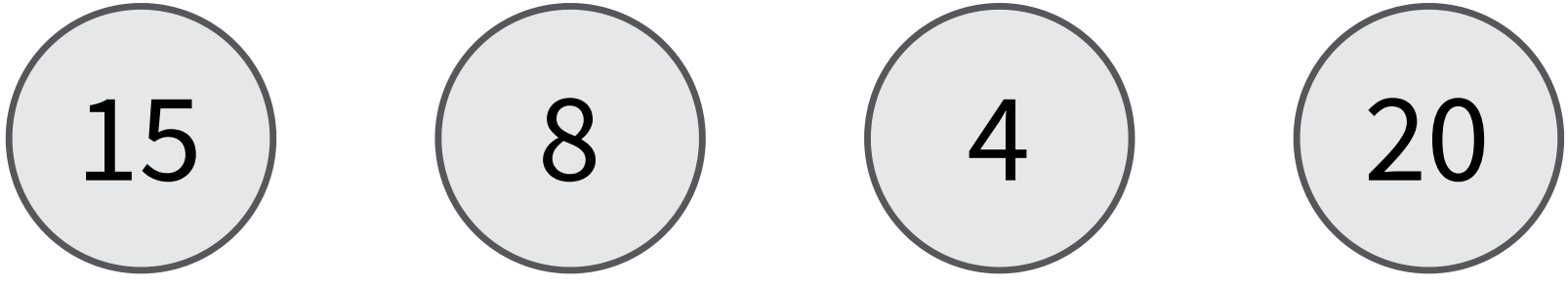
MISE À JOUR — DÉMONSTRATION

- En termes de code, cela pourrait se traduire par les opérations suivantes:

```
// (1) Sélection et association des données
const rectangles = svg.selectAll('rect')
  .data(data[index])
  .style('fill', 'gray');

// (2) Création des nouveaux éléments
const rectanglesCreated = rectangles.enter()
  .append('rect')
  .style('fill', 'green');

// (3) Mise à jour des nouveaux éléments et de ceux existants
rectangles.merge(rectanglesCreated)
  .attr('height', d => d);
```

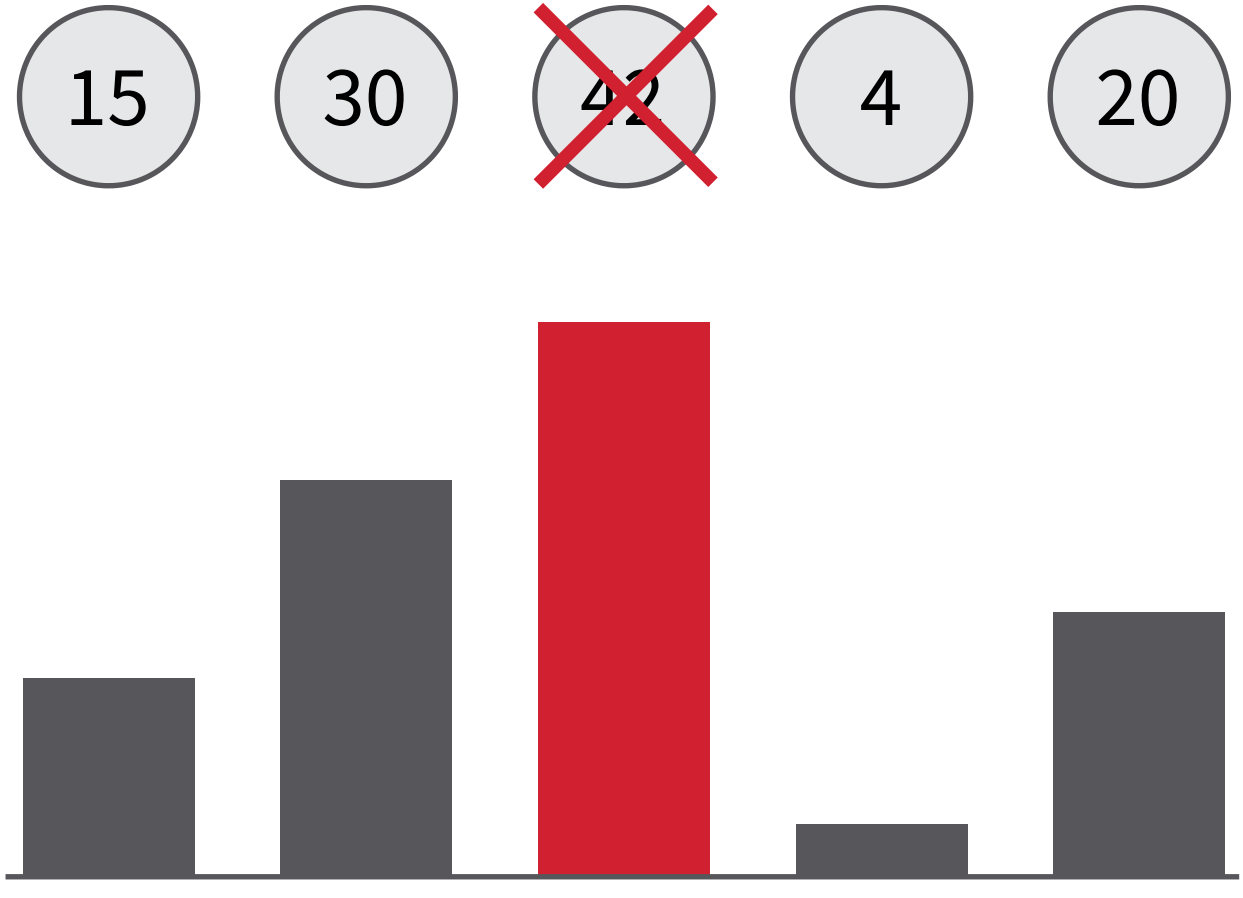
MISE À JOUR — DÉMONSTRATION

- Enfin, supposons que la donnée « **42** » est supprimée de l'ensemble de données à utiliser:



MISE À JOUR — DÉMONSTRATION

- Si une donnée est supprimée de l'ensemble, la barre de trop pourra être supprimée avec la fonction **`exit`**



MISE À JOUR — DÉMONSTRATION

- En termes de code, il faut ajouter la fonction **`exit`** pour supprimer la barre superflue

```
// (1) Sélection et association des données
const rectangles = svg.selectAll('rect')
  .data(data[index])
  .style('fill', 'gray');

// (2) Création des nouveaux éléments
const rectanglesCreated = rectangles.enter()
  .append('rect')
  .style('fill', 'green');

// (3) Mise à jour des nouveaux éléments et de ceux existants
rectangles.merge(rectanglesCreated)
  .attr('height', d => d);

// (4) Suppression des éléments en trop
rectangles.exit()
  .remove();
```

MISE À JOUR — EXEMPLE

```
function update(currentData) {
  // (1) Sélection et association des données
  const rectangles = svg.selectAll('rect')
    .data(currentData)
    .style('fill', 'gray');

  // (2) Création des nouveaux éléments
  const rectanglesCreated = rectangles.enter()
    .append('rect')
    .style('fill', 'green');

  // (3) Mise à jour des nouveaux éléments et de ceux existants
  rectangles.merge(rectanglesCreated)
    .attr('height', d => d);

   // (4) Suppression des éléments en trop
  rectangles.exit()
    .remove();
 }
```

▶ Démo

MISE À JOUR — À NOTER

- À chacune de vos mises à jour, il est important de mettre à jour les **domaines** de vos échelles

MISE À JOUR — POUR EN SAVOIR PLUS

- Exemple concret des mécanismes de mise à jour
- Documentation complète sur les mécanismes de mise à jour

D3.JS — TRANSITIONS

TRANSITIONS

- Les transitions permettent à l'utilisateur de prendre conscience des changements sur les données
- D3.js intègre un mécanisme très **simple** à utiliser pour réaliser des transitions

TRANSITIONS

- Une fois qu'une sélection est effectuée, vous pouvez appeler la fonction **`transition`** de D3.js
- On utilise alors les fonctions **`attr`** ou **`style`** pour réaliser une transition vers les nouvelles valeurs réglées

TRANSITIONS — CONFIGURATION

- `duration` (durée de la transition en ms)
- `delay` (délai avant d'exécuter la transition en ms)
- etc.

TRANSITIONS — EXEMPLE (1)

```
const svg = d3.select('svg');
const circle = svg.append('circle')
  .attr('cx', 50)
  .attr('cy', 50)
  .attr('r', 50)
  .style('fill', 'green');

circle.transition()
  .delay(1000)
  .duration(750)
  .attr('cx', 950)
  .style('fill', 'red');
```

▶ Démo

TRANSITIONS — EXEMPLE (2)

```
function update(currentData) {
  /* ... */

  // (3) Mise à jour des nouveaux éléments et de ceux existant
  rectangles.merge(rectanglesCreated)
      .transition()
      .duration(500)
      .attr('x', d => x(d))
      .attr('y', d => HEIGHT - y(d))
      .attr('width', d => x.bandwidth())
      .attr('height', d => y(d));

  /* ... */
}
```

▶ Démo

TRANSITIONS — POUR EN SAVOIR PLUS

- Documentation complète sur les transitions avec D3.js

D3.JS — MODÈLES DE VISUALISATION

MODÈLES DE VISUALISATION

- D3.js fournit un **très grand nombre** de modèles de visualisation qui peuvent être utilisés
- Dans le cadre de cet atelier, nous nous attarderons sur les plus communs

MODÈLES DE VISUALISATION — TYPES

Arbre et *cluster*

Carte proportionnelle (*treemap*)

Diagramme à cordes (*chord diagram*)

Diagrammes circulaires (*pie charts* et *donut charts*)

Histogramme

Pack

Images tirées de *The Data Visualisation Catalogue*

DIAGRAMMES CIRCULAIRES

- Regroupent les *pie charts* et les *donut charts*
- La fonction **`d3.pies`** doit être employée pour générer les angles associés aux données à afficher
- La fonction **`d3.arcs`** doit être utilisée pour générer les `paths` associés aux arcs de cercle

DIAGRAMMES CIRCULAIRES — EXEMPLE

```
// ...
const arc = d3.arc()
  .innerRadius(0) // Modifier le "innerRadius" pour un donut c
  .outerRadius(radius);

const arcs = d3.pie()(data);

g.selectAll('path')
  .data(arcs)
  .enter()
  .append('path')
  .attr('d', arc)
  .style('fill', (d, i) => color(i));
```

▶ Démo

HISTOGRAMME

- Représente la répartition d'une variable continue selon un certain nombre d'intervalles
- La fonction `d3.histogram` permet de répartir les données selon les bons intervalles

HISTOGRAMME

- Une fois la fonction `d3.histogram` configurée, il faut l'appeler avec les données à diviser
- Cela retournera un tableau de tableaux (`Array[Array]`)
- Chacun des éléments du tableau retourné possèdera les attributs `x0` et `x1`, correspondant au minimum et maximum de l'intervalle

HISTOGRAMME — DÉMONSTRATION

Données

```
data = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Configuration

```
histogram = d3.histogram()
 .domain([0, 9])
 .thresholds(2);
```

Séparation en intervalles

```
histogram(data)
```

```
[ [0, 1, 2, 3, 4],
[5, 6, 7, 8, 9] ]
```

HISTOGRAMME — EXEMPLE

```
// ...
const histogram = d3.histogram()
  .domain(x.domain())
  .thresholds(5);

const bins = histogram(data);
svg.selectAll('rect')
  .data(bins)
  .enter()
  .append('rect')
  .attr('x', d => x(d.x0))
  .attr('y', d => HEIGHT - y(d.length))
  .attr('width', d => x(d.x1) - x(d.x0))
  .attr('height', d => y(d.length));
```

▶ Démo

PACK

- Encode une hiérarchie à l'aide de cercles emboîtés les uns dans les autres
- L'aire d'un cercle représente une certaine quantité
- La fonction **`d3.pack`** permet de générer **facilement** les positions et les rayons des cercles à dessiner

PACK

- Il est possible de configurer la taille d'un *pack* et l'espacement entre les cercles via **`size`** et **`padding`**
- Le pack doit être initialisé à l'aide d'un élément de type **`d3.hierarchy`** (*root*)
- Une fois initialisé, chacun des nœuds du *pack* posséderont les propriétés **`cx`**, **`cy`**, **`r`** et **`data`**.

PACK — EXEMPLE

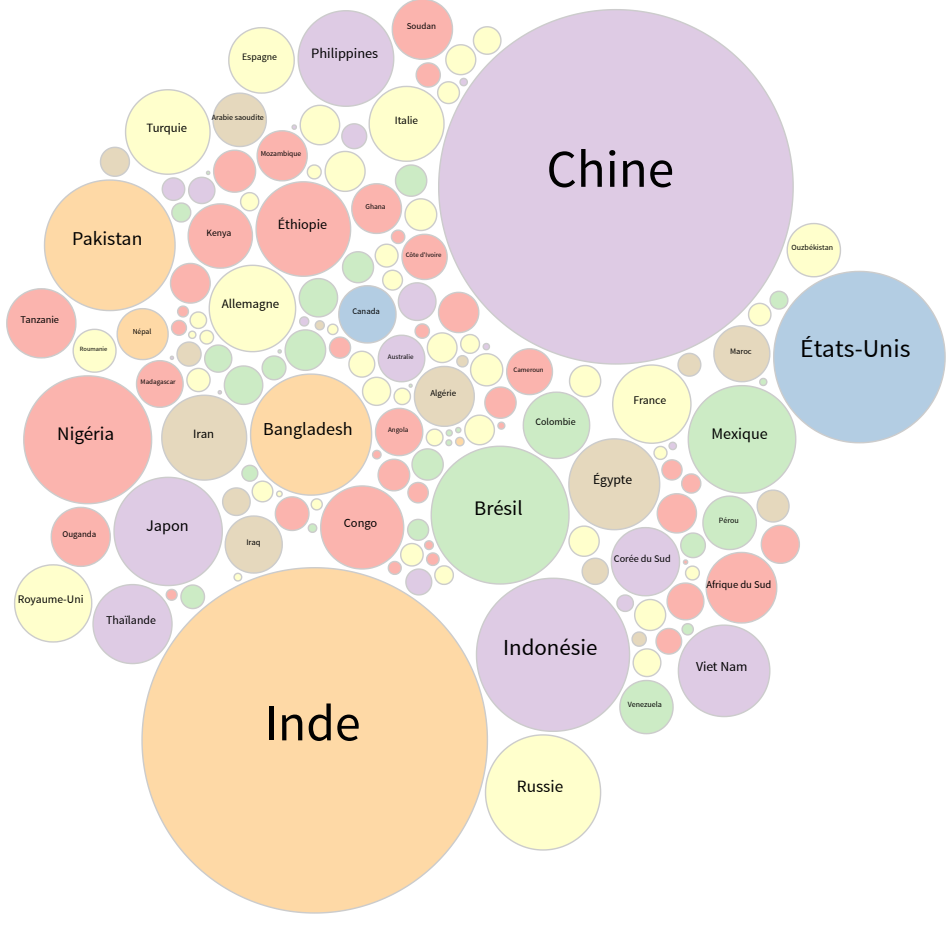
```
// ...
const pack = d3.pack().size([width, height]);
const root = d3.hierarchy(data).sum(d => d.data.population);
const nodes = root.descendants();

pack(root);
svg.selectAll('circle')
  .data(nodes)
  .enter()
  .append('circle')
  .attr('cx', d => d.x)
  .attr('cy', d => d.y)
  .attr('r', d => d.r);
```

Données Démo

MISE EN PRATIQUE

MISE EN PRATIQUE

- Réaliser un *circle packing* à l'aide des données sur la population des pays du monde en 2014.



MISE EN PRATIQUE — DONNÉES

- Les données au format CSV ont été récupérées sur le portail des données ouvertes de la Banque mondiale
- Chaque ligne du fichier CSV indique le **nom** du pays, sa **population** et l'**espérance de vie** de sa population

Données CSV

MISE EN PRATIQUE — DIRECTIVES

- Utiliser les mécanismes de mise à jour pour dessiner les **cercles** et les **textes** en fonction des données
- Colorier les cercles selon leur zone respective
- Dimensionner les textes de manière proportionnelle au rayon du cercle qui leur est associé
- Ajouter une **transition** de **500 ms** pour communiquer un changement dans les données

MISE EN PRATIQUE

- Pour débuter, téléchargez le dossier ZIP contenant le code de départ pour l'exercice
- Complétez, par la suite, le fichier **script.js**
- Utilisez **Firefox** ou un serveur web local pour tester votre script*

Télécharger ▶ Correction

* Google Chrome bloque les requêtes asynchrones réalisées dans un fichier local si celui-ci n'est pas hébergé sur un serveur local