On va découvrir les principes de base de l'API d3.js. Cette API n'est pas limitée au dessin d'images SVG. Ses méthodes peuvent être employées dans d'autres situations. Dans tous les cas, il s'agit de génération et modification d'éléments du DOM.

Créez un nouveau dossier pour ce TD : votre compte / Multimedia / TD3.

1. Mise en page de données simples

On va commencer par le plus simple, afficher une liste d'items.

a. Voici un fichier HTML à télécharger (clic droit sur l'icône ci-contre). Nommez-le `td3_exo11.htm` :

```
<!DOCTYPE html>
<html>
<head>
  <title>TD3 exo 1.1</title>
  <!-- voir https://d3js.org/getting-started -->
  <script src="https://cdn.jsdelivr.net/npm/d3@7"></script>
</head>
<body>
  <ul id="liste"></ul>

  <script>
const aliments = ["chocolat", "brioche", "lait", "confiture", "café", "beurre"]

  </script>
</body>
</html>
```

b. Ouvrez-le avec Visual Studio Code et affichez la prévisualisation temps réel. Affichez la console en prévision.

1.1. Affichage des données

a. Ajoutez les deux lignes suivantes après l'affectation de `aliments` :

```
const liste = d3.select("#liste")
console.log(liste.node())   // selection.node() => élément du DOM
```

- La première ligne effectue une sélection, un seul élément du DOM, celui qui correspond à `#liste`.
- La seconde ligne affiche les éléments HTML qui font partie de la sélection. Ici, c'est la variable `liste` et elle contient seulement l'élément `<ul>`. Commentez cette ligne pour la suite, mais mémorisez la technique pour déboguer.

b. Ajoutez les quatre lignes suivantes :

```
liste.selectAll("li")
  .data(aliments)
  .join("li")
      .text(d => d)
```

C'est nettement plus complexe. Il y a 4 lignes :

- La première ligne repart de la sélection `<ul>` et tente d'y sélectionner des `<li>`. Actuellement il n'y en a aucun, donc cette sélection est vide. Néanmoins, d3.js mémorise l'élément parent de cette sélection, le `<ul>`.
- La deuxième ligne établit une relation entre les (futurs) `<li>` sélectionnés et les données. La méthode `data(valeurs)` associe chaque valeur à un élément du DOM qu'il soit présent ou non dans la sélection.
- La troisième ligne crée les éléments du DOM qui manquent, et s'il y en avait en trop, ça les supprimerait, afin d'avoir exactement le même nombre d'éléments `<li>` que de données. En fait, elle ajuste les éléments aux données.
- La quatrième ligne définit le contenu texte de chaque balise `<li>`. L'écriture `d => d` est assez bizarre. C'est une lambda qui retourne le paramètre qu'on lui passe. Ce paramètre est l'une des valeurs fournies à `data()`, celle qui est associée au `<li>` créé.

La lambda de la quatrième ligne est la seule manière de faire pour que le contenu de chaque balise `<li>` soit spécifique. Essayez de mettre ceci à la place (et remettez comme précédemment après avoir vu le résultat) :

```
      .text('pas très varié')
```

Commentez cette quatrième ligne et essayez successivement :

- une chaîne patron (*template string*) :

```
      .text(d => `Je dois acheter du/de la ${d}`)
```

- ajout d'un attribut, mais pas idéal pour un style :

```
      .text(d => d)
          .attr("style", "list-style-type: circle;")
```

- ajout d'un style (meilleure façon de le faire, voir le cours pour toutes les possibilités) :

```
      .text(d => d)
          .style("list-style-type", "square")
          .style("font-family", "cursive")
```

- imbrication d'un sous-élément :

```
      .append("tt")
          .text(d => d)
```

Pour ce dernier exemple, utilisez l'inspecteur pour comprendre ce qui a été créé : un sous-élément `<tt>` dans chaque `<li>`. La méthode `append("TYPE")` sur une sélection, lui ajoute un sous-élément `<TYPE>`. Et ensuite, c'est lui qui devient la sélection. Notez que la donnée associée a été correctement transmise au sous-élément.

1.2. Données plus complexes

a. Copiez `td3_exo11.htm` en `td3_exo12.htm`. Changez la balise `<title>`.
b. Modifiez l'affectation des données en ceci :

```
const aliments = [
  {nom: "chocolat",  article: "du"},
  {nom: "brioche",   article: "de la"},
  {nom: "lait",      article: "du"},
]
```

c. En reprenant la chaîne patron précédente, modifiez la génération des messages pour que ça affiche, par exemple, « Je dois acheter de la brioche ». Dans tous les cas, `data()` reçoit un tableau dont chaque valeur est fournie aux éléments créés par `join()`.

1.3. Ajout dynamique de nouvelles données

a. Copiez `td3_exo12.htm` en `td3_exo13.htm`. Changez la balise `<title>`.

On va ajouter de nouvelles données, après avoir affiché les premières.

b. Ajoutez ceci tout à la fin du script :

```
aliments.push(
  {nom: "confiture", article: "de la"},
  {nom: "café",      article: "du"},
)

liste.selectAll("li")
    .data(aliments)
    .join("li")
        .text(d => `Et j'y rajoute ...`)    // message tel quel
```

Il y a un bug : les précédents aliments ne s'affichent plus ; toutes les lignes sont identiques.

Pour corriger ce problème, il faut distinguer trois cas, trois sous-sélections produites par `data()` :

- l'ajout d'un nouvel élément `<li>`. Ils sont appelés les éléments *entrants*. On va leur mettre un message spécifique.
- la modification (ou pas) d'un élément existant, appelés *update*. On peut les laisser comme ils sont.
- la suppression des éléments HTML en trop par rapport aux données – ici, il n'y en a pas. Ce sont les éléments *sortants*. On doit les supprimer du DOM.

c. Ce que vous avez mis à la fin, il faut le remplacer par ceci :

```
liste.selectAll("li")
  .data(aliments)
  .join(
    enter => {
```

```
      // enter = sélection des nouveaux éléments, pour les nouvelles données
      enter.append("li").text(d => `Et j'y rajoute ${d.article} ${d.nom}`)
    },
    update => {
      // update = sélection des éléments déjà là qu'il faut évent. modifier
      update //.text(d => `Je dois déjà acheter ${d.article} ${d.nom}`)
    },
    exit => {
      // exit = éléments à supprimer car en trop par rapport aux données
      exit.remove()
    }
  )
```

Vous pouvez décommenter toute la ligne `update.text(d =>...)` pour changer le message.

La méthode `join()` demande soit un nom d'élément et elle gère tout elle-même. Ou alors, on lui donne trois lambda, une pour chaque type de sélection. Les noms *enter*, *update* et *exit* sont des conventions.

Il y a un petit point à remarquer, c'est que les trois lambda ou fonctions doivent toujours être dans l'ordre (*enter*, *update*, *exit*). Amusez-vous à mettre la lambda *update* en premier. Déjà, ça affiche des choses bizarres. Ensuite, si vous utilisez l'inspecteur, vous verrez que la structure des éléments n'est pas bonne, avec des `<li>` dans d'autres `<li>`.

L'ordre de ces trois fonctions ou lambda est important quand on les fournit à `join()`. Si on ne veut pas fournir l'une des lambda, par exemple *update*, il faut la remplacer par `undefined`.

1.4. Modification et suppression de données

a. Copiez `td3_exo13.htm` en `td3_exo14.htm`. Changez la balise `<title>`.

La question se pose, que se passe-t-il quand on fait des changements sur les données ?

b. Mettez ceci tout à la fin : 

```
const aliments2 = [
  {nom: "chocolat",  article: "du"},
  {nom: "beurre",    article: "du"},
  {nom: "miel",      article: "du"},
  {nom: "brioche",   article: "de la"},
]

liste.selectAll("li")
  .data(aliments2)
  .join(
    enter => {
      enter.append("li").text(d => `Et j'y rajoute ${d.article} ${d.nom}`)
    },
    update => {
```

```
      update.text(d => `Je dois toujours acheter ${d.article} ${d.nom}`)
    },
    exit => {
      exit.text(d => `Je ne dois plus acheter ${d.article} ${d.nom}`)
    }
  )
```

On voit tout à la fin que le café, dernier élément de la liste précédente n'est plus à acheter. Par contre, très bizarrement, tous les nouveaux articles sont marqués comme devant toujours être achetés, alors qu'ils ont été rajoutés. Et d'autres n'y sont plus, comme le chocolat, mais ne sont pas signalés.

C'est parce qu'il manque un identifiant pour les données. Sans cet identifiant, les valeurs qu'on fournit à `data()` ne sont vues que pour leur quantité. C'est à dire que `data()` voit passer de 5 à 4 valeurs, et pour ça, il faut seulement garder 4 `<li>` et supprimer le 5e.

Avec un identifiant, `data()` sait exactement ce qui doit changer.

c. Il faut faire un changement dans tous les appels à `data()` depuis le début du script, remplacer :

```
  .data(...)
```

par :

```
  .data(..., d => d.nom)
```

Ça consiste à ajouter une lambda qui, à chaque valeur, retourne quelque chose qui l'identifie de manière unique.

Maintenant, l'énumération affichée est beaucoup plus longue et on voit bien ce qui s'est passé lors des changements de données.

NB: Ici, la lambda *exit* ne supprime pas les anciens `<li>`. Normalement, on doit faire `exit.remove()`.

2. Diagramme très simple

On va maintenant dessiner un diagramme de points (*scatter plot*).

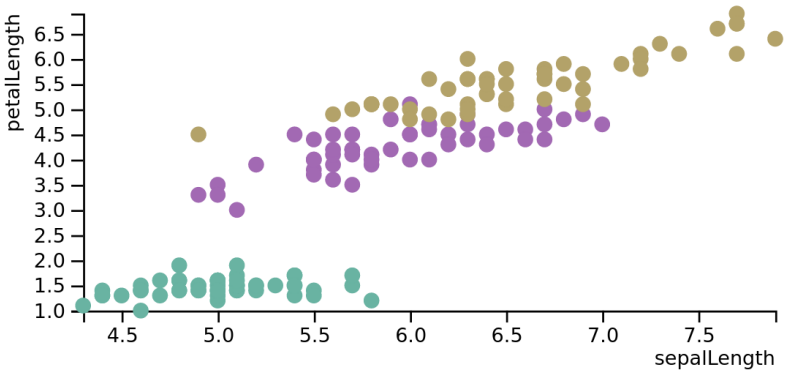

Figure 1: Diagramme de points, version finale

2.1. Préparatifs

a. Téléchargez iris.json. C'est un fichier qui contient des mesures de longueur et largeur des pétales et sépales d'une centaine d'iris de trois espèces différentes.
b. Téléchargez ce fichier HTML. Nommez-le `td3_exo2.htm` :

```
<!DOCTYPE html>
<html>
<head>
  <title>TD3 exo 2</title>
  <!-- voir https://d3js.org/getting-started -->
  <script src="https://cdn.jsdelivr.net/npm/d3@7"></script>
</head>
<body>
  <svg id="dessin"></svg>

  <script>
const width = 400
const height = 200

  </script>
</body>
</html>
```

On va peu à peu construire le script.

2.2. Chargement des données

On commence par lire le fichier `iris.json` et le récupérer en tant que tableau JavaScript.

b. Ajoutez ceci dans le script :

```
function ScatterPlot(iris) {
  console.log(iris)       // à supprimer une fois que c'est ok
```

```
}

d3.json("iris.json")
  .then(iris => ScatterPlot(iris))
```

On va devoir placer les instructions de dessin dans une fonction, `ScatterPlot`, car `d3.json(nomfichier ou URL)` est *asynchrone*. C'est à dire qu'elle « sort » immédiatement mais il y a un traitement qui continue en arrière-plan. C'est le résultat de ce traitement qui nous intéresse, les données à dessiner, donc on est obligé d'attendre. Ça se fait sur une *promesse*, c'est à dire une sorte de réveil pour plus tard quand la fonction aura fini (ou aura planté). La méthode `then(lambda)` spécifie ce qu'il faut faire quand la promesse a réussi. Ici, c'est appeler la fonction de dessin.

Comme `ScatterPlot()` est une fonction à un paramètre qui correspond à la promesse, les données, on peut simplifier l'appel :

```
d3.json("iris.json").then(ScatterPlot)
```

Il existe une autre syntaxe JavaScript pour attendre sur une promesse :

```
async function main() {
  const iris = await d3.json("iris.json")
  ScatterPlot(iris)
}

main()
```

Ici, on définit une fonction `main()` asynchrone, c'est à dire que dedans, il y a une attente sur promesse. Dans cette fonction, il y a un appel à `d3.json()` pour récupérer les données. Lorsque les données arriveront, elles seront mises dans une variable et ensuite on appellera la fonction `ScatterPlot()`.

Vous choisirez la syntaxe qui vous convient. La seconde est un peu plus moderne que la première mais les deux sont équivalentes.

Les promesses sont un mécanisme incroyablement puissant de JavaScript. Ici, on ne fait qu'utiliser une fonction qui retourne une promesse. Il n'est pas prévu dans ce TD de montrer comment créer une telle fonction, mais ce n'est pas très compliqué.

2.3. Dessin des données

On va maintenant construire la fonction `ScatterPlot()`. Ce qui suit est à programmer à l'intérieur de cette fonction.

a. Définissez une constante, `svg` pour contenir la sélection de l'élément `<svg>`.
b. Ajoutez un attribut `viewBox` valant `` `-5 -5 ${width+10} ${height+10}` ``.

C'est à dire :

```
function ScatterPlot(iris) {
  const svg = ... sélection d3.js de l'élément <svg>
  svg.attr("viewBox", `-5 -5...`)
}
```

NB: on fait quelque chose de malpropre : les constantes `width` et `height` sont définies globalement, et on les utilise dans la fonction `ScatterPlot()` sans les passer en paramètre. Dans le TD4, on verra comment transformer tout cela en fermeture et ajouter des *setters* pour le paramétrer.

Pour faire un test rapide, on va dessiner les données de manière très simplifiée.

c. Comme pour les `<ul>` et `<li>` de la première partie, faites en sorte de rajouter un `<circle>` pour chaque iris des données. C'est une structure comprenant un `selectAll()`, un `data()` et un `join()` simple.

d. Il faut paramétrer les cercles de la manière suivante :
   - attribut `cx` : `d => d.sepalLength * 80 - 260`
   - attribut `cy` : `d => d.petalLength * 25`
   - attribut `r` : `width / 100`
   - attribut `fill` : `"red"`

Normalement, vous devez voir apparaître des cercles rouges formant plus ou moins deux lignes. Ces lignes regroupent des iris de la même variété.

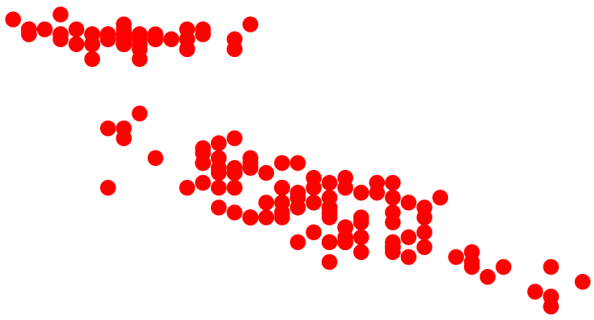

Figure 2: Version 1

Les cercles sont positionnés bizarrement, pas comme dans la figure 1 de la page 5, et on aimerait voir des couleurs pour distinguer les trois variétés d'iris.

2.4. Couleurs par variété

a. Définissez ce tableau au début, après `width` et `height` :

```
const colors = {
  "setosa":     "#69b3a2",
  "versicolor": "#a269b3",
  "virginica":  "#b3a269",
}
```

b. Définissez l'attribut `fill` de chaque cercle par une lambda qui retourne `colors[d.species]` au lieu de `"red"`.

2.5. Placement des points

Les points sont placés par leurs coordonnées `cx` et `cy`, mais on a mis un calcul *ad hoc* (fait juste pour ça, pas généralisable, pas adaptable).

a. Changez la valeur de `width` et de `height` et voyez si le graphique s'adapte.

On va utiliser des échelles (*scale*). Une échelle permet de traduire un espace en un autre. Ici, on veut traduire l'espace des longueurs de sépales en nombre de pixels horizontaux ; et avec une deuxième échelle, on traduit les longueurs de pétales en pixels verticaux. On va construire une échelle linéaire en indiquant l'amplitude des longueurs (*domain*) et celle des pixels (*range*) doc détaillée.

b. Au début de la fonction `ScatterPlot()`, après la définition de la *viewbox*, ajoutez ceci :

```
const fX = d3.scaleLinear()
  .domain(d3.extent(iris, d => d.sepalLength))  // amplitude des données
  .range([0, width])                            // amplitude des pixels

const fY = d3.scaleLinear()
  .domain(d3.extent(iris, d => d.petalLength))  // amplitude des données
  .range([height, 0])                           // amplitude des pixels
```

La méthode `d3.extent(données, quelchamp)` retourne $[min_{données}(champ), max_{données}(champ)]$. Il y a beaucoup de méthodes comme ça décrites dans d3-array#summarize.

NB: dans les tutoriels de Mike Bostock, `fX` et `fY` sont appelées simplement `x` et `y`. Ça peut perturber la compréhension.

Vous avez noté la sorte d'inversion du *range* de `fY`. Pourquoi ?

c. Maintenant remplacez le calcul des attributs `cx` et `cy` par
   - attribut `cx` : `d => fX(d.sepalLength)`
   - attribut `cy` : `d => fY(d.petalLength)`

d. Changez la valeur de `width` et de `height` et voyez si le graphique s'adapte.

2.6. Ajout d'axes

On va ajouter des graduations en bas et à gauche. Ces graduations vont automatiquement s'adapter aux données grâce à `fX` et `fY`.

Avant de rajouter les graduations, il faut faire un peu de place autour du graphique. On va utiliser la *convention des marges*. C'est une configuration qui permet de dessiner un graphique en faisant totalement abstration des marges, et en même temps que bien positionner les titres et les axes qui sont dans ces marges.

a. Au début du script, après `width` et `height`, définissez cette constante :

```
const margin = {top: 20, right: 10, bottom: 30, left: 40}
```

b. Modifiez les échelles. Il faut laisser le *domain* inchangé, et seulement changer le *range* :

```
const fX = d3.scaleLinear()
  ...
  .range([margin.left, width - margin.right])

const fY = d3.scaleLinear()
  ...
  .range([height - margin.bottom, margin.top])
```

c. Remettez la `viewBox` à `` `0 0 ${width} ${height}` ``. La valeur précédente rajoutait des marges en trichant.

d. Maintenant on ajoute les axes. Mettez ceci à la fin de la fonction `ScatterPlot()` :

```
svg.append("g")
    .attr("transform", `translate(0,${height - margin.bottom})`)
    .call(d3.axisBottom(fX))

svg.append("g")
    .attr("transform", `translate(${margin.left},0)`)
    .call(d3.axisLeft(fY))
```

Les axes sont créés par les méthodes `d3.axisBottom(echelle)` et `d3.axisLeft(echelle)`. Ces méthodes ajoutent les lignes, graduations (*ticks*) et numérotations (*labels*). Ces nombreux tracés sont mis dans un groupe qui est translaté selon les marges.

Dans de nombreux graphiques du même genre, on écrit toujours les mêmes instructions pour ajouter des axes. Il y a beaucoup de possibilités de configuration, voir la doc.

2.7. Ajout de titres sur les axes (optionnel)

On aimerait avoir les noms des axes, en bas et à gauche.

a. Créez un élément `<g>` appelé `gTitresAxes` dans `svg` et ajoutez-lui les attributs suivants :
- style `font-family` valant `"sans-serif"`
- attribut `font-size` valant `height/20`
- style `text-anchor` valant `"end"`
- style `fill` valant `"black"`

b. Mettez ceci en place :

```
gTitresAxes.append("text")
    .attr("x", width - margin.right)
    .attr("y", height)
    .attr("dominant-baseline", "ideographic")
    .text("sepalLength")
```

```
gTitresAxes.append("text")
  .attr("transform", "rotate(-90)")
  .attr("x", -margin.top)
  .attr("y", 0)
  .attr("dominant-baseline", "hanging")
  .text("petalLength")
```

Voir cette démonstration pour comprendre l'attribut `dominant-baseline`.

C'est enfin fini !