On va découvrir deux concepts importants de JavaScript, les lambdas et les fermetures. C'est très utilisé dans d3.js pour définir des composants réutilisables.

Créez un nouveau dossier pour ce TD : votre compte / Multimedia / TD4.

1. Point de départ

On repart du diagramme de points du TD3. Voici le squelette du fichier `td3_exo2.htm` obtenu tout à la fin. Téléchargez td3_final.htm :

```
<!DOCTYPE html>
<html>
<head>
  <title>TD3 final</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 = ...
const height = ...
const margin = ...
const colors = {...}

function ScatterPlot(iris) {
  const svg = ...
  svg.attr("viewBox", ...)

  const fX = d3.scaleLinear()...
  const fY = d3.scaleLinear()...

  svg.append("g")...
      .call(d3.axisBottom(fX))
  svg.append("g")...
      .call(d3.axisLeft(fY))

  const gAxes = svg.append("g")...
  gAxes.append("text")...
      .text("sepalLength")
  gAxes.append("text")...
    .text("petalLength")

  svg.selectAll("circle")
    .data(iris)
    .join("circle")...
}
```

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

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

👉 Copiez ce source et appelez-le `td4_exo1.htm`. Changez son titre. On va le modifier tout au long de ce TD.

Le problème de ce code source, est que tout le graphique est reconstruit à chaque fois que les données ou la configuration changent. Et aussi, ce script n'est ni modulaire, ni réutilisable. Mike Bostock propose plusieurs améliorations que ce TD propose de découvrir.

2. Paramétrage du graphique

Actuellement, le programme se présente ainsi :

```
const width = ...
const height = ...
const margin = ...
const colors = {...}

function ScatterPlot(points) {
  // générer le graphique avec les variables globales width, height, etc.
}
```

On voit déjà que la configuration du graphique est sous forme de variables globales. Alors que se passerait-il dans un script plus vaste, où certaines de ces variables serviraient à autre chose et pourraient aussi, par erreur, avoir une valeur concernant d'autres fonctionnalités – si vous copiez la fonction `ScatterPlot()` dans un autre logiciel.

On pourrait envisager de passer toutes ces valeurs en paramètre de la fonction `ScatterPlot()` :

```
function ScatterPlot(width, height, margin, colors, points) {
  // générer le graphique avec les paramètres width, height, etc.
}

ScatterPlot(400, 200, {...}, {...}, iris)
```

Ce n'est pas une bonne solution générale. Si on veut rendre le graphique réutilisable, on sera amenés à ajouter de très nombreux paramètres de configuration (titres...).

Mike Bostock propose pour commencer, de grouper les paramètres essentiels du graphique dans un seul objet :

```
function ScatterPlot(config, points) {
  // générer le graphique avec config.width, config.height, etc.
}

const config = {
  // identifiant de l'élément <svg> concerné
  idsvg:  "#dessin",

  // dimensions du graphique
  width: 400,
  height: 200,
  ...
}

ScatterPlot(config, iris)
```

👉 Copiez `td4_exo1.htm` en `td4_exo2.htm` et faites cette transformation sur `td4_exo2.htm`. Attention, toutes les anciennes variables, `width`, `height`, etc doivent être remplacées par `config.width`, `config.height`, etc. Il y a aussi l'identifiant de l'élément `<svg>`.

Dans la sous-fonction `dessiner`, on peut renommer les données `iris` en `points`, parce qu'on va aller vers davantage de polyvalence.

3. Fonctions d'accès

Pour continuer à rendre la fonction de dessin réutilisable, on peut aussi définir deux fonctions d'accès aux données, l'une pour les abscisses, l'autre pour les ordonnées. Dans le TD3, c'étaient les *lambdas* `d => d.sepalLength` et `d => petalLength`. Cherchez les deux endroits où elles étaient employées. L'un est évident, l'autre est un peu plus subtil à trouver.

👉 Copiez `td4_exo2.htm` en `td4_exo3.htm` et ajoutez ceci : 

```
const config = {
  width: 400,
  height: 200,
  ...

  titleX: "Sepal Length",
  getX: d => d.sepalLength,

  titleY: "Petal Length",
  getY: d => d.petalLength,
}

ScatterPlot(config, iris)
```

On a maintenant deux titres et deux *lambdas*, `getX` et `getY`, qu'il faut employer dans la fonction `ScatterPlot()`.

👉 Faites en sorte d'utiliser `config.titleX` et `config.titleY`.

👉 Faites en sorte d'utiliser `config.getX` et `config.getY`.

- Le premier endroit à modifier est dans le domaine des deux échelles, `fX` et `fY`. Les *lambda* concernées sont clairement visibles ; mettez directement `config.getX` et `config.getY` à leur place. En effet, avant, il y avait `d => calcul sur d`. Maintenant, on y met directement la fonction `config.get` qui fait ce calcul.
- Le second endroit à modifier est dans la génération des points pour chaque donnée. On voit par exemple `d => fX(d.sepalLength)`. Il faut remplacer par `d => fX(config.getX(d))`, on ne peut pas mettre `fX(config.getX)`. Pourquoi ? Idem pour y.

👉 Il y a aussi les couleurs des iris. Actuellement, la couleur est placée lors du dessin, il y a un accès à `colors[d.species]`. Il faudrait plutôt ajouter une ligne dans `config` : `getColor: ...` et faire en sorte que le dessin l'utilise.

Le problème avec cette façon de faire, c'est qu'alors, on ne voit plus trop la différence entre les variables qui contiennent de simples valeurs et celles qui contiennent des *lambda*. L'écriture `d3.extent(iris, config.getX)` est très compacte, mais masque le fait que `config.getX` est une fonction.

Pour terminer sur les *lambda*, il existe une autre syntaxe pour les écrire :

```
const getX = d => d.sepalLength
const getX = function(d) { return d.sepalLength }
```

Ces deux écritures sont à peu près équivalentes, voir fonctions fléchées et opérateur function. La différence est que la variable `this` n'est pas définie dans une *lambda*.

4. Fermetures

Cette solution d'un objet de configuration est intéressante, mais on doit gérer la fonction de dessin avec sa configuration. Ça peut être compliqué s'il y a plusieurs graphiques à dessiner avec des configurations différentes. Par exemple, on a dessiné le graphe des longueurs de sépales et de pétales, et on pourrait en dessiner un autre avec les largeurs des pétales et sépales, et encore un autre avec les longueurs et les largeurs des pétales, etc. Donc à chaque fois, il faut redéfinir un nouvel objet config, et ne pas se tromper d'objet à fournir aux fonctions.

Donc Mike Bostock propose de lier la configuration à la fonction, de faire une sorte d'objet qui contient à la fois la configuration et la fonction de dessin. Ainsi, il n'y aura plus qu'une seule chose à gérer, représentant le graphique entier avec sa configuration. Cela se fait avec une *fermeture* (*closure* en anglais). Voir la fin du CM2.

4.1. Fermeture simple

Le principe est de programmer une fonction $f1$, qui contient une fonction $f2$, et qu'elle retourne en tant que résultat. L'idée est d'appeler d'abord la fonction $f1$, puis d'appeler son résultat ultérieurement.

👉 Téléchargez fermeture.htm :

```
<html>
<head>
<script>
function f1() {
  var message = "bonjour"

  function f2() {
    console.log(`je suis f2, message=${message}`)
  }

  console.log("je suis f1 et je retourne la fonction f2")
  return f2
}

const F = f1()
console.log(`F vaut ${F}`)
F()
</script>
</head>
</body>
```

👉 Ouvrez-le dans le navigateur, avec la console pour voir les messages.

La fonction `f2` mise dans la variable `F` est une fermeture. Elle enferme la variable[1] `message` avec elle. Cette variable lui est extérieure, comme si elle était globale. C'est une variable locale de `f1`. Quand on quitte `f1`, normalement, les variables locales sont supprimées... sauf celles qui sont enfermées dans des fermetures.

Les fermetures sont une alternative à la définition d'une classe. Elles permettent de définir une sorte d'objet ayant des propriétés et capable d'être appelé, ce qui n'est pas possible avec une classe (sauf en Python). Les fermetures peuvent être directement appelées en tant que fonctions et avoir des variables membres comme les instances de classes. Au contraire, les instances d'une classe ne peuvent pas être considérées comme des fonctions (sauf avec cette proposition).

Par contre, les fermetures ne peuvent pas hériter d'une autre fermeture. On ne peut pas créer une hiérarchie de fermetures, comme on le fait pour les classes.

4.2. Modification d'une fermeture

👉 Ajoutez ceci à la fin de `fermeture.htm` et testez : 

```
console.log(`F.message = ${F.message}`)
F.message = "bonsoir"
F()
```

[1] Le mot clé `var` permet de déclarer une variable visible de la totalité de la fonction. Les mots clés `let` et `const` déclarent des variables/constantes visibles uniquement dans le bloc `{...}` où elles sont créées. Ici, ça ne changerait rien de déclarer `message` avec `let`.

Il ne se passe rien de bien. La variable `message` n'est ni accessible, ni modifiable directement. On va procéder autrement.

👉 Ajoutez ceci dans `f1`, juste avant le `return f2` :

```
f2.getMessage = () => message
f2.setMessage = (m) => message = m
```

puis ceci à la fin du script :

```
console.log(`F.getMessage() = ${F.getMessage()}`)
F.setMessage("bonne nuit")
F()
```

On a ajouté un *getter* et un *setter* à la fermeture. Notez qu'ils doivent passer par `f2`, comme si la variable enfermée, `message`, était déjà membre de `f2`.

👉 Faites un dernier essai : remplacez la variable message de `f1` par un paramètre, avec une valeur par défaut :

```
function f1(message="salut") {

  // variable globale à la fonction
  // remplacée par le paramètre:  var message = "bonjour"
```

Cela permet de définir le message initial, si on veut :

```
const Fb = f1()
Fb()
const Fc = f1("hello")
Fc()
```

4.3. Paramétrage d'une fermeture

Voici encore une variante pour découvrir un mécanisme très puissant dans JavaScript, le passage d'un objet en paramètre et sa séparation en différentes variables.

👉 Ajoutez ceci à la fin :

```
function g1({auteur, message, date}) {

  function g2() {
    console.log(`message=${message}, daté de ${date}, signé ${auteur}`)
  }

  return g2
}

const G = g1({date: "hier", message: "à demain", auteur: "Pierre"})
G()
```

Les paramètres de `g1` sont déclarés comme étant un objet avec plusieurs propriétés, auteur, message et date. On doit donc fournir un objet comme ça à l'appel de `g1`. Cet objet fest décomposé en trois variables, `auteur`, `message` et `date`. C'est ce qu'on appelle une *décomposition d'objet* en JavaScript.

Il y a une variante très utile à connaître. Elle permet de définir des valeurs par défaut.

👉 Ajoutez ceci à la fin : 

```
function h1({auteur, message="bonjour", date="aujourd'hui"} = {}) {

  function h2() {
    console.log(`autre message=${message}, daté de ${date}, signé ${auteur}`)
  }

  h2.getMessage = () => message
  h2.setMessage = (m) => message = m

  h2.getDate = () => date
  h2.setDate = (d) => date = d

  return h2
}

const Ha = h1({auteur: "Pierre"})
Ha()

const Hb = h1({auteur: "Yannick", date: "demain"})
Hb()

const Hc = h1({auteur: "Pierre", date: "hier", message: "à demain"})
Hc()
```

Regardez les paramètres de `h1`. C'est une *décomposition des paramètres avec des valeurs par défaut.* C'est à dire que, comme pour `g1` plus haut, les paramètres sont déclarés dans une sorte d'objet (ou de tableau), mais avec avec des affectations de valeurs par défaut `{nom1=val1, nom2=val2, ...}` et on termine par `= {}`, une affectation avec un objet vide. C'est expliqué ici en anglais et ici en français.

À l'appel, on fournit un objet qui contient les champs qu'on veut affecter. Ceux qui manquent recevront la valeur par défaut. S'il n'y a pas de valeur par défaut, la variable sera *undefined*. Cette syntaxe permet de définir des valeurs par défaut utilisables dans une fermeture.

4.4. Chaînage des *setters*

Il reste un problème avec les *setters*, on voudrait les chaîner :

```
const Hd = h1({auteur: "Yannick"}).setDate("hiziv").setMessage("demat")
Hd()
```

Pour que les *setters* puissent être chaînés, il faut que chacun retourne la fermeture.

```
h2.setMessage = function (m) {
  message = m
  return h2
}
```

Faites la modification sur les deux *setters* et vérifiez que ça fonctionne.

5. Transformation du graphique en fermeture

Voici le mécanisme pour transformer un graphique en fermeture. L'objet de configuration devient un ensemble de paramètres par défaut. Ces paramètres sont emprisonnés dans une fermeture. Ensuite, dans un second temps, on appelle cette fermeture en lui fournissant les données à dessiner.

```
function ScatterPlot({width=400, height=200, ...}={}) {

  function dessiner(points) {
    // utiliser width, height, etc.
  }

  // getters et setters pour les paramètres utiles

  return dessiner
}

const SC = ScatterPlot({width: 600, height: 300, ...})

// dessiner le graphique avec les données des iris
SC(iris)
```

👉 Copiez `td4_exo3.htm` en `td4_exo4.htm` et effectuez la transformation en fermeture. Voici le début :

```
function ScatterPlot({
    idsvg,
    titleX, getX, titleY, getY, getColor, // titres, accès aux données
    width, height,                        // dimensions
    margin={top: 20, ...}                 // marges avec des valeurs par défaut
}={}) {

  function dessiner(iris) {
```

```
    // élément `<svg>` global
    const svg = d3.select(idsvg)

    ...
  }

  return dessiner
}
```

NB: dans la sous-fonction `dessiner()`, il faut enlever tous les « `config.` » (ou tout recopier de `td4_exo1.htm`) parce que les paramètres sont à nouveau des variables distinctes.

Le lancement, tout à la fin, se fait maintenant ainsi : 

```
const couleurs = {...}

const SC = ScatterPlot({
  idsvg: "#dessin",
  titleX: "Sepal Length",
  getX: d => d.sepalLength,
  titleY: "Petal Length",
  getY: d => d.petalLength,
  getColor: d => colors[d.species],
  // utilisation des valeurs par défaut pour les autres paramètres
})

d3.json("iris.json").then(SC)  // ou la variante async
```

☛ (optionnel) Ajoutez les *getters* et *setters* pour les propriétés définies par défaut : `width`, `height`...

Que pensez-vous de cette manière de faire ? Un peu compliquée, mais beaucoup plus souple. On peut créer une bibliothèque de graphiques configurables et adaptés à nos besoins.

6. Mise à la « norme d3.js »

☛ Copiez `td4_exo4.htm` en `td4_exo5.htm`

Dans la « philosophie d3.js », il serait souhaitable que le graphique soit dessiné ainsi : 

```
d3.json("iris.json")
  .then(iris =>
      d3.select("#dessin")
        .datum(iris)
        .call(SC)
  )
```

C'est à dire que :

1) La sélection de l'élément `<svg id="dessin">` soit faite en amont et non pas par le dessin lui-même. Ça fait référence à la ligne `const svg = d3.select(idsvg)` au début de la fonction `dessiner()` : il faudrait faire autrement. Il faut seulement déplacer `svg` en tant que premier paramètre de la fonction `dessiner()`.
2) Les données sont passées à la fermeture par la méthode `datum(valeurs)`. Cette méthode est toute simple, elle transmet les valeurs à toutes les sélections sur laquelle on l'appelle, c'est à dire ici, l'élément `<svg>`.
3) La fonction de dessin est dans une fermeture, `SC` comme précédemment, mais cette fonction reçoit la sélection en paramètre au lieu du jeu de données.

👉 Pour cela :

a. Dans les paramètres de `ScatterPlot`, enlevez le paramètre `idsvg`.
b. Enlevez la propriété `idsvg` de l'objet de configuration passé à l'appel de `ScatterPlot`.
c. Renommez et modifiez la fonction `dessiner(iris)` en `_dessiner(svg, iris)`. Ça sera une fonction privée[2].
d. Supprimez sa variable locale `svg` puisqu'elle est passée en paramètre.
e. Avant les *getters* et *setters*, mettez la nouvelle définition de `dessiner` :

```
function dessiner(selection) {
  selection.each(function(datum) {
    _dessiner(d3.select(this), datum)   // this = élément <svg>, datum = iris
  })
}
```

La méthode `selection.each(f)` appelle la fonction `f` sur chaque élément de la sélection, en fournissant le paramètre passé préalablement à `datum`, et en affectant `this` avec l'élément DOM de la sélection. NB: Il ne faut pas que la fonction `f` soit une *lambda* sinon `this` n'aurait pas de valeur. Vous voyez comment écrire une fonction qui n'a pas de nom et qui n'est pas non plus une lambda.

7. Mise à jour du tracé

Il reste un dernier point à voir, comment mettre à jour le graphique quand les données changent. Voici comment faire changer les données :

👉 Copiez `td4_exo5.htm` en `td4_exo6.htm` et mettez ceci à la fin :

```
d3.json("iris.json")
  .then(iris => {
      d3.select("#dessin")
        .datum(iris)
        .call(SC)
      // redessiner une petite partie des données
      d3.select("#dessin")
```

[2]On peut rendre une méthode privée dans une classe, en mettant un `#` au début de son nom, mais on ne peut pas faire ça pour une fonction interne.

```
        .datum(iris.slice(0, 30))
        .call(SC)
    }
  )
```

Ça dessine toutes les valeurs, puis ça n'en dessine qu'une partie. Inspectez l'élément `<svg>`. Vous allez constater la présence de deux groupes en double, ce sont ceux des axes et graduations.

La solution est de ne pas ajouter ces groupes s'ils y sont déjà.

👉 Ajoutez ceci au tout début, avant la fonction `ScatterPlot` :

```
d3.selection.prototype.appendIfAbsent = function(selector, type) {
  const element = this.select(selector)
  return element.empty() ? this.append(type) : element
}
```

Ça met en place un ajout à l'API *d3-selection* qui permet de n'ajouter un élément que s'il est absent. En JS, comme en Python, on peut ajouter ses propres fonctions à des bibliothèques existantes.

👉 Partout où il y a `svg.append(TYPE)`, mettez `svg.appendIfAbsent("#ID", TYPE).attr("id", "ID")`. Le but est de ne pas recréer des éléments singletons comme les axes et les titres. Cela fait 5 éléments concernés, les groupes des axes et celui des titres, et les deux titres. Choisissez les identifiants correctement.