

# JavaScript Canvas

# Table des matières

<b>I. Contexte</b>	<b>3</b>
<b>II. Introduction à l'élément &lt;canvas&gt; et son API</b>	<b>4</b>
A. L'élément <canvas> en HTML.....	4
B. API Canvas en JavaScript.....	5
C. Exercice : Quiz.....	8
<b>III. Affichage d'un graphique statistique</b>	<b>9</b>
A. Introduction.....	9
B. Initialisation .....	9
C. Affichage des axes .....	10
D. Génération des points.....	13
E. Disposition des points.....	14
F. Exercice : Quiz .....	15
<b>IV. Essentiel</b>	<b>16</b>
<b>V. Auto-évaluation</b>	<b>17</b>
A. Exercice .....	17
B. Test.....	17
<b>Solutions des exercices</b>	<b>18</b>

## I. Contexte

**Durée :** 60 min

### Prérequis :

- Navigateur internet
- Base du langage HTML, JavaScript
- Connaissance norme ES6

### Environnement de travail :

- Navigateur internet de votre choix
- Replit : espace de travail HTML

#### Contexte

JavaScript est un langage de programmation polyvalent utilisé pour créer des applications interactives sur le Web. Il est souvent utilisé en conjonction avec le langage de balisage HTML pour rendre les pages web plus dynamiques et interactives. L'un des éléments les plus puissants et intéressants qu'il permet de manipuler est le `<canvas>`.

Le `<canvas>` est un élément graphique HTML qui permet de dessiner sur votre page web des formes complexes et personnalisées grâce à du code JavaScript. `<canvas>` peut être décrit comme une zone de dessin dont la hauteur et la largeur sont définies dans le code HTML. Grâce au JavaScript, on peut accéder à cette zone de dessin à l'aide de différentes fonctions. Une API Canvas nous est fournie pour pouvoir interagir avec cet élément en JavaScript. Ainsi, le `<canvas>` est utilisé pour accomplir en Web des jeux, des animations, ainsi que des graphiques géométriques et statistiques.

#### Attention

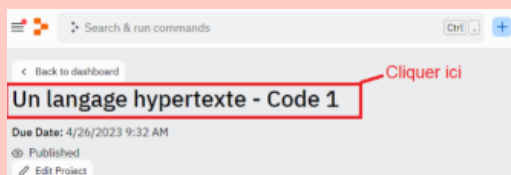
Pour avoir accès au code et à l'IDE intégré de cette leçon, vous devez :

- 1) Vous connecter à votre compte sur <https://replit.com/> (ou créer gratuitement votre compte)
- 2) Rejoindre la Team Code Studi du module via ce lien : <https://replit.com/teams/join/mmurvlgippxuasordloklllbqskoim-programmer-avec-javascript>

Une fois ces étapes effectuées, nous vous conseillons de rafraîchir votre navigateur si le code ne s'affiche pas.

En cas de problème, redémarrez votre navigateur et vérifiez que vous avez bien accepté les cookies de connexion nécessaires avant de recommencer la procédure.

Pour accéder au code dans votre cours, cliquez sur le nom du lien Replit dans la fenêtre. Par exemple :



## II. Introduction à l'élément <canvas> et son API

### A. L'élément <canvas> en HTML

#### Définition

Tout d'abord, il est important de savoir que le <canvas> est un élément HTML de la catégorie « *embedded content* » ou « *contenu intégré* » en français. Cela signifie que cet élément est utilisé pour importer ou insérer une ressource dans votre document HTML, d'autres exemples d'éléments de type « *Contenu intégré* » sont <img>, <audio> ou <video>.

#### Conseil

Si vous souhaitez en savoir plus sur les différentes catégories de contenu des éléments HTML, n'hésitez pas à vous rendre sur MDN Web Docs<sup>1</sup> pour vous renseigner, ou à faire une recherche avec les mots-clés « *content categories HTML* » ou « *catégories de contenu HTML* ».

#### Exemple

```
1 <canvas id="example" title="Example of a canvas element" width="1280px" height="720px">
  </canvas>
```

[cf. ]

Ci-dessus, on peut voir l'exemple d'un usage basique de l'élément <canvas>.

Celui-ci possède donc les attributs communs à tous les éléments HTML comme id, title, ou bien lang.

Mais il possède spécifiquement en addition 2 attributs :

- width : la largeur en pixel de l'élément
- height : la hauteur en pixel de l'élément

#### Remarque

Il est important de noter que <canvas> doit impérativement posséder une balise fermante </canvas> en addition, contrairement à ses confrères de la catégorie contenu de flux comme <img> ou <audio>.

La raison de la présence d'une balise fermante est à cause du contenu « *fallback* ».

La notion de contenu « *fallback* » en informatique correspond à une solution alternative, notamment dans notre contexte, à un contenu qui remplacera le <canvas> si celui-ci ne marche pas ou n'est pas compatible avec le navigateur.

Ici, le contenu « *fallback* » à insérer dans les balises <canvas> peut être du simple texte ou un autre élément HTML, comme démontré dans les 2 exemples ci-dessous.

#### Exemple

```
1 <canvas id="example" title="Example of a canvas element" width="1280px" height="720px">Canvas
  non compatible avec votre navigateur</canvas>
```

[cf. ]

```
1 <canvas id="example" title="Example of a canvas element" width="1280px" height="720px">
2   
3 </canvas>
```

[cf. ]

<sup>1</sup> [https://developer.mozilla.org/fr/docs/Web/HTML/Content\\_categories](https://developer.mozilla.org/fr/docs/Web/HTML/Content_categories)

## B. API Canvas en JavaScript

Tout d'abord, avant d'en apprendre plus sur l'API Canvas, il ne faut pas oublier les méthodes JavaScript que fournissent l'élément <canvas>.

En dehors des méthodes communes à tous les éléments HTML, les méthodes JS que le <canvas> possède sont au nombre de 4 :

- `getContext(contextId, options)` : renvoie le contexte de dessin du Canvas, ce qui correspond à son API pour dessiner. Le paramètre `contextId` permet de spécifier l'API désiré (« 2d », « bitmaprenderer », « webgl », « webgl2 », or « webgpu »), le paramètre `options` est un objet non obligatoire.
- `toDataURL(type, quality)` : renvoie l'URL de l'image du Canvas. Les 2 paramètres sont optionnels, le premier est le `type` de l'image (par défaut : PNG) et le deuxième, `quality`, est la qualité désirée en décimale entre 0 et 1.
- `toBlob(callback, type, quality)` : fonctionne comme `toDataURL`, mais au lieu d'une URL il renvoie un objet Blob correspondant au fichier de l'image du Canvas. Il possède 1 paramètre en plus qui est une fonction `callback` qui sera appelée après avoir généré le Blob.
- `transferControlToOffscreen()` : méthode avancée négligeable dans ce contexte, renvoie un nouvel objet Canvas dit « offscreen » qui ne dépend plus du DOM.

L'API du <canvas> est récupérable en JavaScript grâce à la méthode `getContext()` de cet élément.

Comme dit précédemment, en fonction du paramètre `type` de cette méthode, nous pouvons récupérer plusieurs API différentes, par exemple avec le type « webgl », nous obtiendrons logiquement l'API WebGL pour dessiner sur le Canvas.

Néanmoins, celle qui nous intéresse sera juste l'API du type « 2d » qui permet simplement de dessiner des formes en 2d.

### Exemple

Voici en exemple ci-dessous l'usage de la méthode `getContext()` pour récupérer l'API :

```
1 const canvasElement = document.getElementById('example');  
2 const ctx = canvasElement.getContext('2d');
```

[cf. ]

L'API Canvas possède de nombreuses méthodes qui sont répartissables en plusieurs catégories, néanmoins nous allons juste nous concentrer sur les notions essentielles et basiques pour dessiner sur le Canvas.

Une de ces catégories de méthode, qui est sans aucun doute la plus essentielle et basique, est celle des « Path ».

Un « Path » dans l'API correspond à un chemin, mais plus précisément à une liste de points et de vecteurs, qui forment ce fameux chemin.

Voici les méthodes principales qui permettent de manipuler les « Path » de l'API :

- `beginPath()` : ouvre un nouveau chemin.
- `moveTo(x, y)` : ajoute un nouveau point au chemin sans le connecter au précédent point.
- `lineTo(x, y)` : ajoute un nouveau point au chemin en le connectant au précédent point sous forme de ligne. (Crée donc un vecteur qui connecte le point précédent et le nouveau point).
- `closePath()` : connecte le premier et le dernier point en ligne.
- `stroke()` : dessine les contours des chemins (soit les connexions entre les points).
- `fill()` : dessine l'intérieur des chemins (soit l'espace créé par la connexion de plusieurs points).

Dessiner une ligne avec un Path correspond donc à ouvrir un chemin à un point A, et ajouter un vecteur jusqu'à un point B.

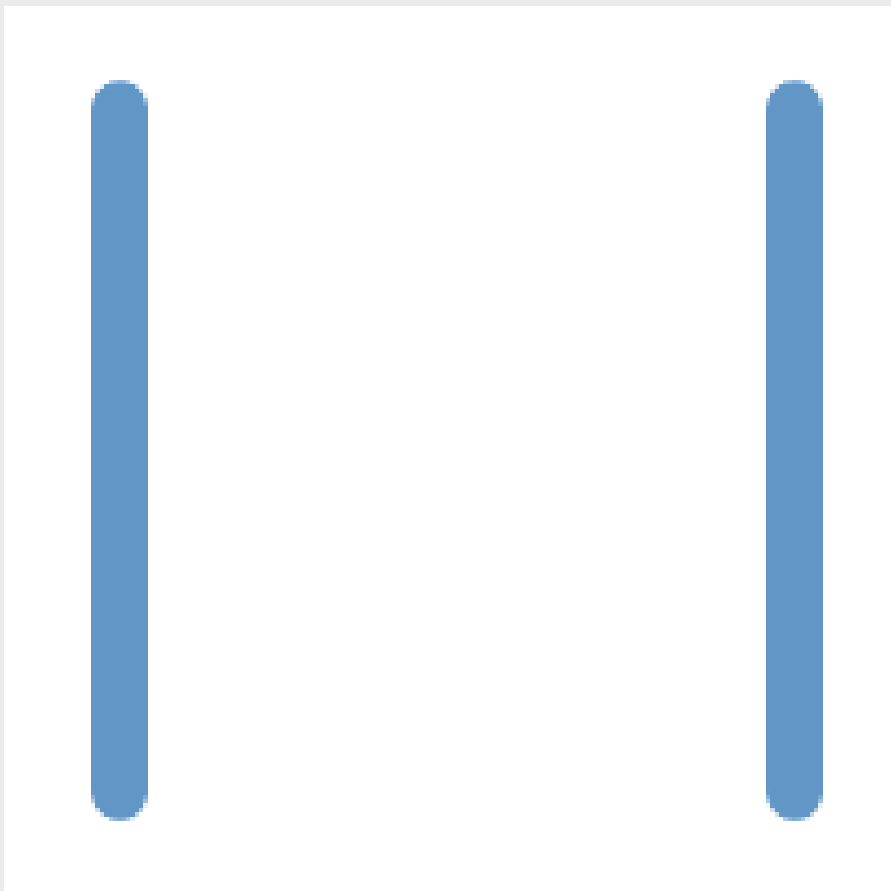
Voyons avec un exemple : comment nous pouvons dessiner 2 lignes parallèles, ainsi qu'un triangle.

#### Exemple

```
1 ctx.beginPath();
2 ctx.moveTo(75, 50);
3 ctx.lineTo(75, 75);
4 ctx.moveTo(100, 50);
5 ctx.lineTo(100, 75);
6 ctx.stroke();
```

[cf. ]

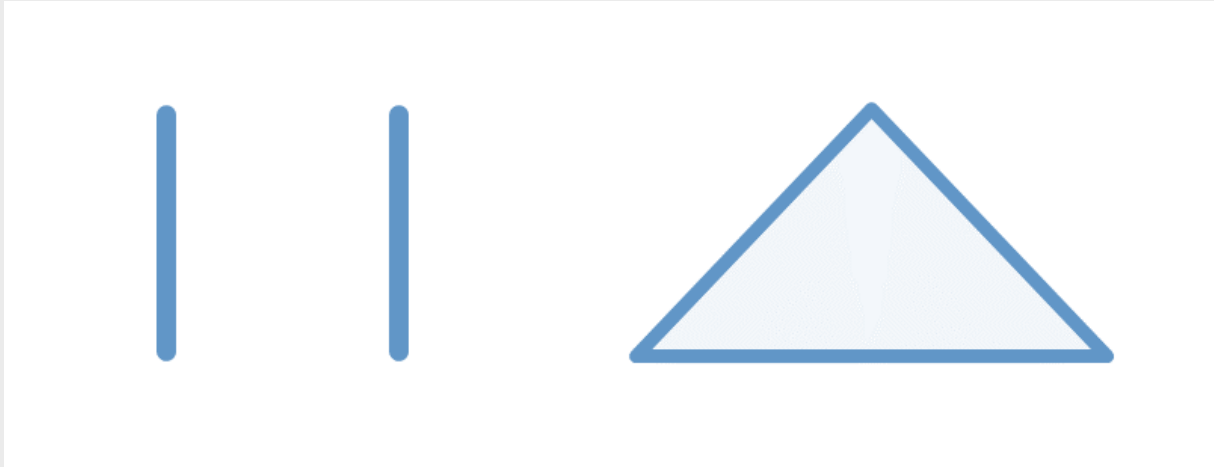
Ce code permet de dessiner simplement 2 lignes parallèles. Ainsi, nous obtenons le résultat visuel suivant :



```
1 ctx.moveTo(150, 50);
2 ctx.lineTo(175, 75);
3 ctx.lineTo(125, 75);
4 ctx.closePath();
5 ctx.stroke();
```

[cf. ]

Si nous ajoutons le code ci-dessus, un triangle sera dessiné à côté des lignes parallèles :



#### Conseil

N'hésitez pas à manipuler les valeurs et à utiliser par vous-même les méthodes vues précédemment pour bien comprendre leurs fonctionnements.

Nous venons de voir les méthodes pour dessiner manuellement les contours des formes dans le Canvas. Néanmoins, si nous souhaitons réaliser des formes complexes, se limiter aux méthodes `moveTo()` et `lineTo()` risque de rendre la tâche ardue et fastidieuse.

Bien heureusement, au lieu de se contenter de connecter des points en ligne droite, l'API Canvas 2D possède aussi certaines méthodes afin de réaliser des formes plus spécifiques :

- `fillRect()`, `strokeRect()`, `clearRect()` : ces méthodes permettent respectivement de dessiner un rectangle rempli, les contours d'un rectangle, et d'effacer une partie du Canvas sous forme de rectangle. Elles prennent toutes les mêmes paramètres `x`, `y`, `width` et `height` qui correspondent aux coordonnées ainsi qu'aux dimensions du rectangle.
- `rect()` : fonctionne comme les méthodes précédentes, mais au lieu de dessiner un rectangle, l'ajoute sous forme de chemin au Path actuel.
- `fillText()`, `strokeText()` : respectivement dessine du texte et dessine les contours du texte grâce aux paramètres `text`, `x` et `y` qui correspondent respectivement au texte et à ses coordonnées.
- `arc()` : ajoute un cercle ou un arc sous forme de chemin au Path actuel grâce aux paramètres suivants : `x`, `y`, `radius`, `startAngle`, `endAngle` qui correspondent respectivement aux coordonnées du centre, au rayon du cercle, et les mesures de l'angle de l'arc en radians (pour un cercle, `startAngle` et `endAngle` doivent donc avoir une différence de  $2 * \pi$ ). Cette méthode possède aussi un 5<sup>e</sup> paramètre optionnel, `counterclockwise`, qui détermine la direction de dessin de l'arc.

#### Complément

Il existe d'autres méthodes permettant de dessiner des formes dans l'API, comme `quadraticCurveTo()`, `bezierCurveTo()` ou `arcTo()` qui reposent sur des concepts géométriques complexes.

Néanmoins, ces méthodes étant assez avancées et non essentielles, elles ne relèvent pas de notre contexte d'apprentissage.

La dernière notion essentielle de l'API est celle du **style** de cette dernière.

Grâce à certaines variables attributs qu'elle contient, nous pouvons styliser le trait de dessin de l'API en les modifiants :

- `strokeStyle` : couleur du contour du dessin.
- `fillStyle` : couleur de remplissage du dessin.
- `shadowOffsetX` : distance de décalage de l'ombre du trait en X.
- `shadowOffsetY` : distance de décalage de l'ombre du trait en Y.
- `shadowBlur` : niveau de flou de l'ombre.
- `shadowColor` : couleur de l'ombre.
- `font` : paramètre de la police de texte.

#### Attention

Si vous souhaitez modifier le style du dessin sans affecter ce qui a été précédemment dessiné, il est impératif de commencer un nouveau Path avec la méthode `beginPath()`, car le style s'appliquera à l'entièreté du Path actuel.

#### Conseil

Si vous souhaitez vous renseigner plus en détail sur l'API, n'hésitez pas à vous rendre sur MDN Web Docs<sup>1</sup> ou à faire des recherches avec les mots-clés « *CanvasRenderingContext2D API* ».

## C. Exercice : Quiz

[solution n°1 p.19]

### Question 1

L'élément <canvas> est de la catégorie « *embedded content* » : quel est donc son usage ?

- ☐ Définir l'en-tête de sa section
- ☐ Interagir avec l'utilisateur
- ☐ Importer ou insérer une ressource

### Question 2

L'élément <canvas> peut s'écrire sans balise fermante.

- ☐ Vrai
- ☐ Faux

### Question 3

Quelle méthode permet de récupérer l'API Canvas ?

- ☐ `getContext2D()`
- ☐ `getAPI()`
- ☐ `getContext()`
- ☐ `getAPI2D()`

<sup>1</sup> <https://developer.mozilla.org/fr/docs/Web/API/CanvasRenderingContext2D>



## Question 4

La méthode `closePath()` permet de fermer un Path pour en débiter un nouveau.

- ☐ Vrai
- ☐ Faux

## Question 5

Laquelle de ces méthodes ne dessine pas un rectangle ?

- ☐ `fillRect()`
- ☐ `strokeRect()`
- ☐ `rect()`

### III. Affichage d'un graphique statistique

#### A. Introduction

Maintenant que nous avons passé en revue toutes les notions essentielles de l'API Canvas, nous allons mettre en pratique ces connaissances nouvellement acquises et réaliser un graphique de nuage de points avec le `<canvas>`.

Pour coder, nous allons nous servir de l'outil web Replit, rendez-vous donc sur le lien suivant : [replit<sup>1</sup>](https://replit.com/languages/html), afin de créer un nouvel espace de travail HTML.

#### B. Initialisation

##### Méthode

Nous allons tout d'abord mettre en place les fondations. En créant un espace de travail avec le template HTML de Replit, par défaut 3 fichiers ont été créés : « `index.html` », « `script.js` » et « `style.css` ».

Dirigeons-nous donc dans le fichier « `index.html` », et modifions-y le `<body>` afin d'y ajouter le `<canvas>` et de charger « `script.js` ».

```
1 <body>
2   <canvas id="graph" width="1000px" height="500px">Canvas not supported</canvas>
3   <script src="script.js"></script>
4 </body>
```

##### Attention

Il est important que le script soit chargé après le `<canvas>`, autrement lorsque nous souhaiterions récupérer le `<canvas>` dans le script, celui-ci ne sera pas encore existant.

##### Méthode

Désormais, nous ne devrions plus avoir besoin d'interagir avec le code HTML, il ne nous reste plus qu'à récupérer le `<canvas>` et son API en JavaScript. Donc dans le fichier « `script.js` », mettons en place le code suivant :

```
1 const canvasElement = document.getElementById('graph');
2 const ctx = canvasElement.getContext('2d');
```

<sup>1</sup> <https://replit.com/languages/html>

### Conseil

Il est considéré comme une bonne pratique en Web d'utiliser l'outil « *inspecter* » de votre navigateur sur le résultat afin de vérifier le bon fonctionnement du code mise en place.

## C. Affichage des axes

### Méthode

La première partie de l'algorithme pour réaliser un graphique statistique est de créer les axes de ce dernier.

Créons donc 2 fonctions jumelles qui seront vides pour l'instant, mais que nous appellerons `createAxeX()` et `createAxeY()`.

Ces 2 fonctions prendront les mêmes paramètres suivants qui permettront de personnaliser les axes :

- `marks` : nombre de mesure à afficher sur l'axe,
- `scale` : échelle de valeurs des mesures,
- `unit` : unité de mesure de l'axe (optionnel),
- `label` : nom de l'axe (optionnel).

Voici donc à quoi devraient ressembler les 2 fonctions avec tous ces éléments :

```
1 const createAxeX = (marks, scale, unit, label) => {
2
3 }
```

Concentrons-nous maintenant sur l'axe des abscisses, donc sur la méthode `createAxeX()`.

Nous allons en premier lieu dessiner en un trait l'axe, qui fera toute la largeur du Canvas avec une marge de 50 px. Pour connaître les dimensions du Canvas, l'API contient les attributs `ctx.canvas.width` et `ctx.canvas.height`.

```
1 ctx.beginPath();
2 ctx.moveTo(50, ctx.canvas.height - 50);
3 ctx.lineTo(ctx.canvas.width - 50, ctx.canvas.height - 50);
4 ctx.stroke();
```

### Attention

Pour observer le résultat, n'oubliez pas d'appeler dans le script la méthode `createAxeX()` après l'avoir définie.

### Méthode

Maintenant que cet axe s'affiche, ajoutons-y son label. Étant un paramètre optionnel, il faut donc placer ce code dans un `if` conditionnel.

```
1 if (label) {
2     ctx.textAlign = "center";
3     ctx.fillText(label, ctx.canvas.width / 2, ctx.canvas.height - 10);
4 }
```

Vous pouvez tester le bon fonctionnement en ajoutant des paramètres factices lors de l'appel de la fonction comme dans l'exemple ci-dessous.

```
1 createAxeX(0, 0, undefined, "Income");
```

[cf. ]

**Remarque**

Dans le code précédent, nous utilisons l'attribut de l'API `ctx.textAlign` qui permet de définir l'alignement du texte, c'est-à-dire si celui-ci doit être centré, aligné à gauche ou aligné à droite.

**Méthode**

Il ne nous reste plus qu'à afficher les mesures sur l'axe. Le nombre des mesures à placer est défini par le paramètre `marks`, tandis que l'échelle de chacune de ces mesures est définie par `scale`.

Nous allons donc nous servir d'une boucle `for` et d'un peu de logique mathématique pour décider de comment placer ces mesures.

Tout d'abord, voici la boucle qui nous permettra d'itérer pour le nombre de mesures à placer :

```
1 for (var i = 0; i <= marks; i += 1) {
2
3 }
```

Dans cette boucle, ajoutons un calcul qui va nous permettre de définir la distance en pixel entre chaque mesure. Cette distance est calculée grâce à la taille de l'axe, soit `ctx.canvas.width - 100` (le 100 correspond à la marge en pixel), taille que nous divisons par le nombre de mesures :

```
1 const distance = (ctx.canvas.width - 100) / marks;
```

En dessous du calcul de la distance, nous pouvons directement ajouter le code qui dessinera le trait des mesures :

```
1 ctx.beginPath();
2 ctx.moveTo(50 + i * distance, ctx.canvas.height - 50);
3 ctx.lineTo(50 + i * distance, ctx.canvas.height - 45);
4 ctx.stroke();
```

Il ne nous reste plus qu'à afficher la valeur de la mesure en texte, sans oublier qu'il y a l'unité de la mesure à prendre en compte, étant optionnelle.

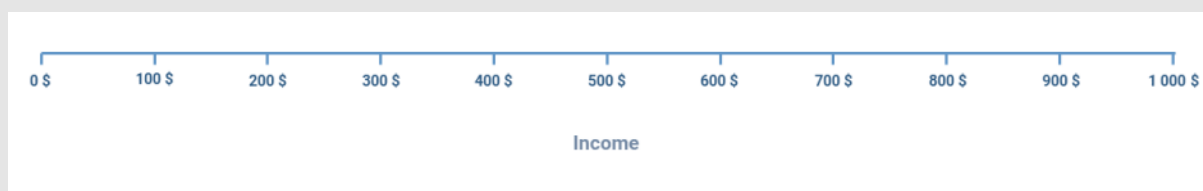
```
1 ctx.textAlign = "center";
2 if (unit) ctx.fillText(scale * i + unit, 50 + i * distance, ctx.canvas.height - 35);
3 else ctx.fillText(scale * i, 50 + i * distance, ctx.canvas.height - 35);
```

Il ne nous reste plus qu'à essayer cette méthode en l'appelant ainsi avec ces paramètres :

```
1 createAxeX(10, 100, "$", "Income");
```

[cf.]

Vous devriez ainsi obtenir le résultat visuel suivant :

**Méthode**

Avant de passer à la suite, il nous faut aussi reproduire la même fonction, mais pour l'axe Y.

Le concept sera le même, néanmoins il n'existe pas de méthode dans l'API Canvas qui permette d'écrire du texte verticalement, ce qui est nécessaire étant donné que l'axe Y est vertical. Voici donc une fonction que vous pouvez ajouter. En l'utilisant, vous pourrez écrire verticalement exactement de la même manière que la méthode `fillText()`.

```

1 const fillTextVertically = (text, x, y) => {
2   const lineHeight = 10;
3   y = y - lineHeight * text.length / 2;
4
5
6   ctx.textAlign = "center";
7   for (var i = 0; i < text.length; i += 1) {
8     ctx.fillText(text[i], x, y + i * lineHeight);
9   }
10 }

```

Vous possédez désormais tous les éléments pour réaliser la méthode `createAxeY()` par vous-même, ce à quoi nous vous encourageons afin de mieux prendre en main l'API Canvas.

Voici le résultat que vous devriez obtenir en adaptant la fonction pour l'axe Y :

```

1 const createAxeY = (marks, scale, unit, label) => {
2   ctx.beginPath();
3   ctx.moveTo(50, 50);
4   ctx.lineTo(50, ctx.canvas.height - 50);
5   ctx.stroke();
6
7
8   if (label) {
9     ctx.textAlign = "center";
10    fillTextVertically(label, 10, ctx.canvas.height / 2);
11  }
12
13
14  for (var i = 0; i <= marks; i += 1) {
15    const distance = (ctx.canvas.height - 100) / marks;
16
17
18    ctx.beginPath();
19    ctx.moveTo(50, ctx.canvas.height - 50 - i * distance);
20    ctx.lineTo(45, ctx.canvas.height - 50 - i * distance);
21    ctx.stroke();
22
23
24    ctx.textAlign = "center";
25    if (unit) ctx.fillText(scale * i + unit, 35, ctx.canvas.height - 50 - i * distance);
26    else ctx.fillText(scale * i, 35, ctx.canvas.height - 50 - i * distance);
27  }
28 }

```

Essayons maintenant d'appeler les méthodes de création des axes avec les paramètres ci-dessous :

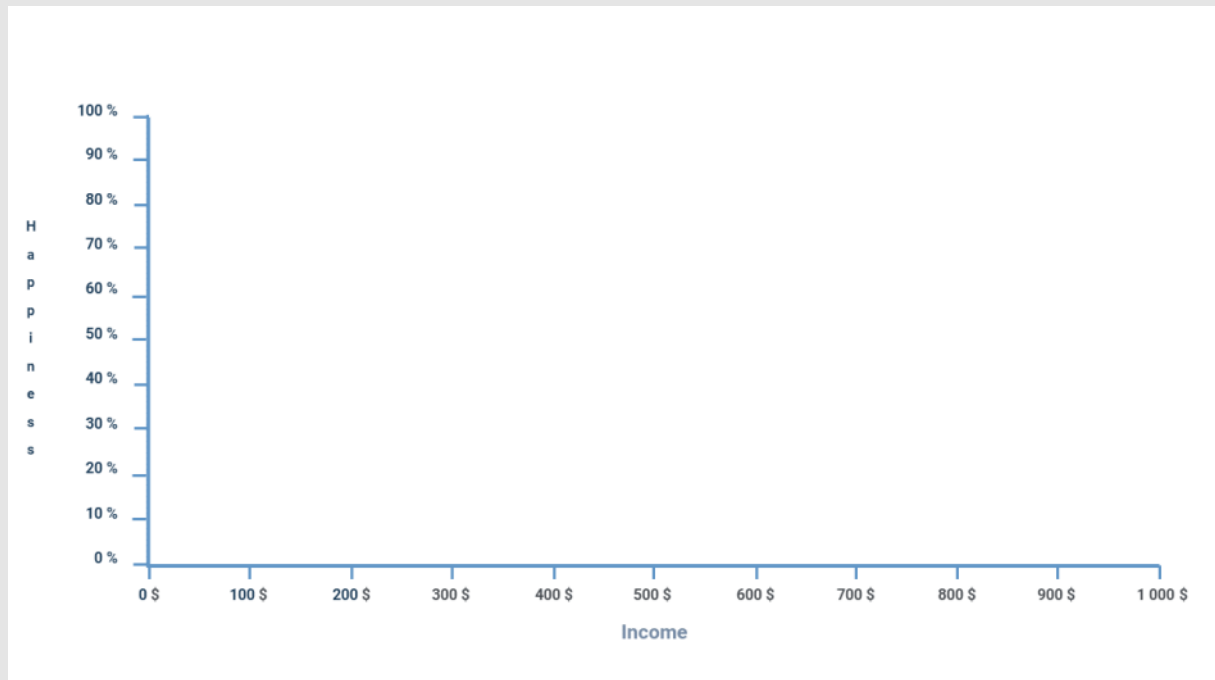
```

1 createAxeX(10, 100, "$", "Income");
2 createAxeY(10, 10, "%", "Happiness");

```

[cf. ]

Ainsi, le résultat obtenu devrait visuellement correspondre à cet exemple :



## D. Génération des points

### Méthode

Nous avons réussi à afficher les axes du graphique, il ne nous reste donc plus qu'à placer des points dessus.

Mais pour l'instant, nous n'avons toujours pas de données qui correspondent à des points à placer, nous allons donc nous occuper d'implémenter une fonction utilitaire générant aléatoirement ces points.

Nommons cette fonction `generatePoints()`, son premier paramètre est `size`, soit la taille du tableau de point généré, suivi de `x_min`, `x_max`, `y_min`, `y_max` qui correspondent à la délimitation des coordonnées des points, et finalement `color`, qui sera la couleur du point lorsque nous afficherons.

Voici donc ce à quoi la définition de la méthode correspond en suivant les consignes précédentes :

```
1 const generatePoints = (size, x_min, x_max, y_min, y_max, color) => {
2
3 }
```

Nous souhaitons créer un tableau de points, et itérer avec le paramètre `size` pour ajouter le bon nombre de points dans le tableau.

Initialisons la variable qui jouera le rôle du tableau, et créons la boucle `for` où nous générerons le point à chaque boucle :

```
1 const points = [];
2
3
4 for (var i = 0; i < size; i += 1) {
5
6
7 }
8
9
```

```
10 return points;
```

Il ne nous reste plus qu'à ajouter dans la boucle un point avec des coordonnées aléatoire et la couleur attribuée en paramètre.

Pour générer aléatoirement les coordonnées, servons-nous de la fonction `Math.random()` de JavaScript, qui nous renvoie un nombre décimal aléatoire entre 0 et 1.

Voici le code permettant d'ajouter un point en adaptant la fonction `Math.random()` afin de correspondre aux délimitations fournies en paramètres :

```
1 points.push({
2   x: Math.random() * (x_max - x_min) + x_min,
3   y: Math.random() * (y_max - y_min) + y_min,
4   color: color
5 });
```

Avec cela, la fonction est complète, il ne nous reste plus qu'à l'appeler afin de récupérer les points générés dans une variable.

```
1 const points = generatePoints(50, 0, 1000, 50, 100, "red");
2 console.log(points);
```

[cf. ]

## E. Disposition des points

### Méthode

Nous possédons tout le nécessaire pour placer les points sur le graphique, nous allons maintenant pouvoir finalement créer la fonction qui s'en occupera.

Appelons cette fonction `placePoints()`, celle-ci prendra en paramètre `points`, le tableau des points, et `marks_x`, `scale_x`, `marks_y`, `scale_y` qui sont les mêmes valeurs données lors de la création des axes, et dont nous avons besoin pour placer correctement nos points à l'échelle.

Voici le prototype de fonction correspondant aux consignes :

```
1 const placePoints = (points, marks_x, scale_x, marks_y, scale_y) => {
2
3 }
```

Par ergonomie, au sein de la fonction nous allons créer 2 variables `scale_x` et `scale_y` qui seront les échelles dont nous avons besoin pour adapter les coordonnées natives au graphique.

La valeur maximale de chaque axe est calculable en multipliant le nombre de mesures `marks` et l'échelle des mesures `scale` ensemble, et en divisant cette valeur par la taille en pixel de l'axe, nous obtenons l'échelle pour transformer la coordonnée native en coordonnées adaptées au graphique.

```
1 const transform_x = marks_x * scale_x / (ctx.canvas.width - 100);
2 const transform_y = marks_y * scale_y / (ctx.canvas.height - 100);
```

Nous allons ensuite ajouter une boucle « *forEach* » qui va itérer à travers chaque point du tableau, ce sera donc la boucle pour placer les points.

La première chose que nous devons accomplir dans cette boucle va être de calculer les coordonnées mises à l'échelle. Nous allons donc créer en même temps que la boucle 2 variables en son sein qui représenteront les coordonnées transformées.

```
1 points.forEach(point => {
2   const x = 0;
3   const y = 0;
4 });
```

En prenant en compte la marge, l'inversion de l'axe Y et la mise à l'échelle avec les variables `transform_x` et `transform_y`, le résultat des coordonnées adaptées au graphique devrait correspondre au calcul suivant :

```
1 const x = 50 + point.x / transform_x;
2 const y = ctx.canvas.height - 50 - point.y / transform_y;
```

Utilisons ces coordonnées avec la méthode `arc()` que nous avons précédemment vue, ainsi nous pouvons ajouter le code suivant qui dessinera sur le graphique les points à chaque itération :

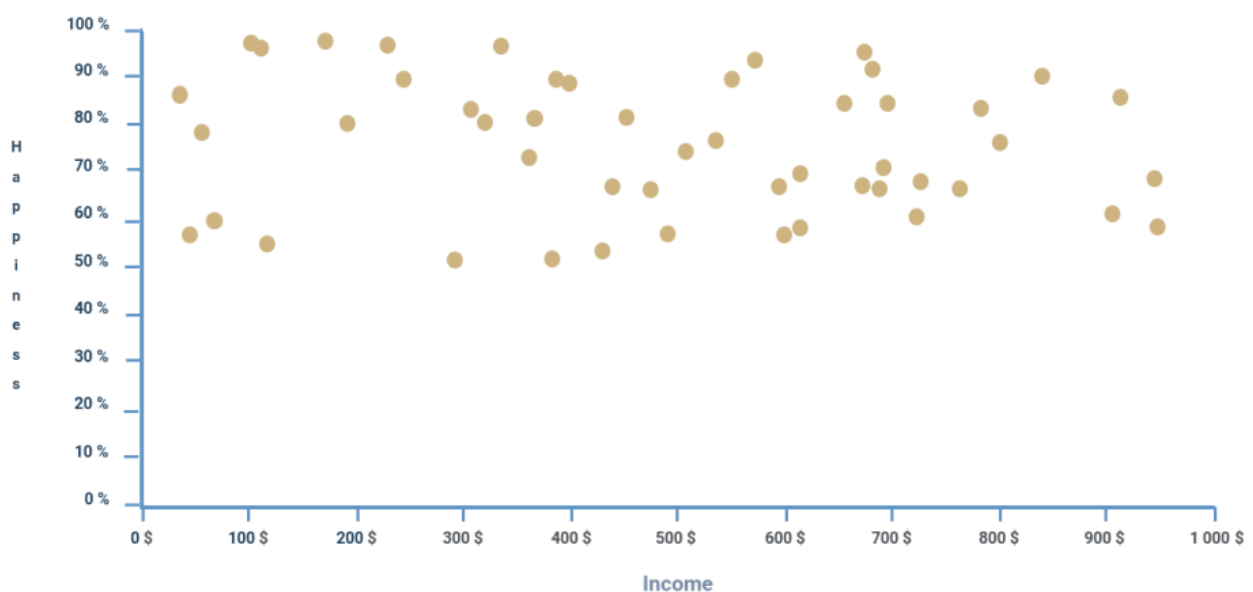
```
1 ctx.beginPath();
2 ctx.arc(x, y, 5, 0, 2 * Math.PI);
3 ctx.fillStyle = point.color;
4 ctx.fill();
```

Enfin, voici la ligne que nous allons taper pour afficher nos points :

```
1 placePoints(points, 10, 100, 10, 10);
```

[cf. ]

Les valeurs étant aléatoires, il serait complexe de vérifier correctement le bon fonctionnement de la disposition des points. Néanmoins, le code peut être considéré valide si le résultat est semblable à celui-ci dessous :



#### Remarque

N'oubliez pas que la disposition des points dépend des limitations fournies en paramètres à la fonction `generatePoints()`.

## F. Exercice : Quiz

[solution n°2 p.20]

### Question 1

Quel est le type de l'API Canvas que nous utilisons ?

- ☐ « 2d »
- ☐ « *webgl2* »
- ☐ « *webgpu* »
- ☐ « *bitmaprendering* »

Question 2

Dans la méthode `createAxeX()`, laquelle des déclarations suivantes dessine la ligne de l'axe ?

- ☐ `ctx.moveTo(50, ctx.canvas.height - 50);`
- ☐ `ctx.lineTo(ctx.canvas.width - 50, ctx.canvas.height - 50);`
- ☐ `ctx.stroke();`

Question 3

`Math.random()` peut générer la valeur 500.

- ☐ Vrai
- ☐ Faux

Question 4

Laquelle des affirmations suivantes concernant `generatePoints(50, 0, 1 000, 50, 100, « red »)` est fausse ?

- ☐ 50 points seront générés
- ☐ La coordonnée X des points sera une valeur aléatoire entre 0 et 1 000
- ☐ La coordonnée Y des points sera une valeur aléatoire entre 0 et 100
- ☐ Ces points seront dessinés en rouge

Question 5

Dans la méthode `placePoints()`, quelle valeur correspond à l'échelle pour calculer la coordonnée X adaptée au graphique ?

- ☐ `scale_x`
- ☐ `generatePoints()`
- ☐ `transform_x`

## IV. Essentiel

Dans le cadre de notre cours, nous avons exploré l'utilisation de Canvas, un élément HTML qui permet de créer des graphiques, des images et des animations à l'aide de JavaScript. Canvas offre une grande flexibilité et nous permet de manipuler le contenu et le comportement du canevas en utilisant diverses méthodes JavaScript.

L'une des premières étapes pour travailler avec Canvas consiste à obtenir le contexte de rendu du canevas en utilisant la méthode `getContext()`. Cette méthode est utilisée pour spécifier si nous souhaitons travailler en 2D avec le contexte « 2d » ou en 3D avec le contexte « *webgl* ». Une fois que nous avons obtenu le contexte de rendu, nous pouvons commencer à dessiner sur le canevas. Par exemple, nous pouvons utiliser la méthode `fillRect(x, y, largeur, hauteur)` pour dessiner un rectangle rempli sur le canevas, en spécifiant les coordonnées x et y



du coin supérieur gauche du rectangle, ainsi que sa largeur et sa hauteur. Une autre méthode utile est `strokeRect(x, y, largeur, hauteur)`, qui trace uniquement le contour d'un rectangle sans le remplir. Pour effacer une partie spécifique du canevas, nous pouvons utiliser la méthode `clearRect(x, y, largeur, hauteur)`.

En utilisant ces méthodes, nous pouvons créer des formes simples sur le canevas. Cependant, Canvas offre beaucoup plus de possibilités, comme le dessin de courbes, de lignes, de textes, d'images et même d'animations. En combinant les différentes méthodes de dessin et en manipulant les propriétés du canevas, nous pouvons créer des expériences interactives et visuellement attrayantes.

## V. Auto-évaluation

### A. Exercice

Vous avez un projet important à rendre pour votre école : il s'agit d'un graphique linéaire pour afficher des données stockées en JSON, à réaliser en HTML/JavaScript avec l'API Canvas. Bien heureusement, un de vos précédents projets était de réaliser un nuage de points avec données aléatoires, et vous avez l'autorisation de réadapter le code utilisé.

#### Question 1

[solution n°3 p.21]

En dehors de l'initialisation, quelles seront les différentes étapes de votre algorithme pour réaliser le graphique linéaire ?

#### Question 2

[solution n°4 p.21]

Comment comptez-vous réadapter votre précédent graphique nuage de points pour en faire un graphique linéaire avec l'API Canvas ?

### B. Test

#### Exercice 1 : Quiz

[solution n°5 p.21]

##### Question 1

Lequel de ces attributs l'élément `<canvas>` ne possède-t-il pas ?

- ☐ title
- ☐ width
- ☐ size

##### Question 2

La méthode `toDataURL()` de l'élément `<canvas>` permet d'obtenir le lien de l'image contenu.

- ☐ Vrai
- ☐ Faux

##### Question 3

La méthode `arc()` permet de former un arc avec les mesures de l'angle de début et l'angle de fin en radian. Quelles valeurs faut-il renseigner pour que ces 2 paramètres forment un cercle complet ?

- ☐ 0, Pi
- ☐ 0, 2\*Pi
- ☐ 0, 360
- ☐ - 180, 180

#### Question 4

La déclaration `Math.random() * (x_max - x_min)` génère une valeur aléatoire en `x_max` et `x_min`.

- ☐ Vrai
- ☐ Faux

#### Question 5


Dans la fonction `placePoints()`, quelle déclaration obtient la coordonnée finale en Y d'un point ?

- ☐ `50 + point.y / transform_y`
- ☐ `point.y / transform_y`
- ☐ `ctx.canvas.height - 50 - point.y / transform_y;`

### Solutions des exercices


**Exercice p. 8 Solution n°1****Question 1**

L'élément `<canvas>` est de la catégorie « *embedded content* » : quel est donc son usage ?

- ☐ Définir l'en-tête de sa section
- ☐ Interagir avec l'utilisateur
- ☒ Importer ou insérer une ressource
-  En HTML, les « *embedded content* » ou « *contenu intégré* » sont utilisés pour importer une ressource ou intégrer du contenu externe.


**Question 2**

L'élément `<canvas>` peut s'écrire sans balise fermante.

- ☐ Vrai
- ☒ Faux
-  L'élément `<canvas>` nécessite une balise fermante, car elle doit contenir un contenu ; « *fallback* » ou « *solution alternative* » sera affiché si le `<canvas>` n'est pas supporté.


**Question 3**

Quelle méthode permet de récupérer l'API Canvas ?

- ☐ `getContext2D()`
- ☐ `getAPI()`
- ☒ `getContext()`
- ☐ `getAPI2D()`
-  La méthode `getContext()` de l'élément `<canvas>` permet de récupérer son API en lui donnant en paramètre le type de l'API désiré.

**Question 4**


La méthode `closePath()` permet de fermer un Path pour en débiter un nouveau.

- ☐ Vrai
- ☒ Faux
-  `closePath()` connecte le premier et dernier point du Path, il ne le ferme que graphiquement parlant et non pas littéralement, et n'en débute donc pas un nouveau.

**Question 5**

Laquelle de ces méthodes ne dessine pas un rectangle ?

- ☐ `fillRect()`
- ☐ `strokeRect()`
- ☒ `rect()`


 `rect()` ajoute un rectangle sous forme de chemin au Path actuel, mais ne le dessine pas. Les autres méthodes dessinent un rectangle, mais ne l'ajoutent pas au Path actuel.

## Exercice p. 15 Solution n°2

### Question 1

Quel est le type de l'API Canvas que nous utilisons ?


- ☒ « 2d »
- ☐ « *webgl2* »
- ☐ « *webgpu* »
- ☐ « *bitmaprenderer* »

 Lors de l'utilisation de la méthode `getContext()`, nous fournissons en paramètre la string « 2d » qui permet de récupérer l'API du même type.

### Question 2

Dans la méthode `createAxeX()`, laquelle des déclarations suivantes dessine la ligne de l'axe ?


- ☐ `ctx.moveTo(50, ctx.canvas.height - 50);`
- ☐ `ctx.lineTo(ctx.canvas.width - 50, ctx.canvas.height - 50);`
- ☒ `ctx.stroke();`

 La méthode `moveTo()` ajoute un point au Path sans connexion, la méthode `lineTo()` ajoute un point au Path en le connectant au point précédent, et la méthode `stroke()` est celle qui dessine les contours des connexions du Path.

### Question 3

`Math.random()` peut générer la valeur 500.


- ☐ Vrai
- ☒ Faux

 `Math.random()` ne peut générer qu'une valeur décimale entre 0 et 1, il faut se servir de cette valeur dans un calcul pour obtenir un nombre aléatoire au-delà de cet éventail.

### Question 4

Laquelle des affirmations suivantes concernant `generatePoints(50, 0, 1 000, 50, 100, « red »)` est fausse ?

- ☐ 50 points seront générés
- ☐ La coordonnée X des points sera une valeur aléatoire entre 0 et 1 000
- ☒ La coordonnée Y des points sera une valeur aléatoire entre 0 et 100
- ☐ Ces points seront dessinés en rouge

 Dans les paramètres de la fonction, `y_min` et `y_max`, qui correspondent aux délimitations de l'éventail des valeurs aléatoires générées en Y, sont mises respectivement à 50 et 100.


**Question 5**

Dans la méthode `placePoints()`, quelle valeur correspond à l'échelle pour calculer la coordonnée X adaptée au graphique ?

☐ `scale_x`

☐ `generatePoints()`

☒ `transform_x`

 `scale_x` correspond à l'échelle d'une mesure sur l'axe, elle est utilisée pour calculer `transform_x` qui est l'échelle pour adapter une coordonnée au graphique.

**p. 17 Solution n°3**

Tout d'abord, il faut charger les données du JSON, et les adapter pour pouvoir les utiliser dans l'algorithme. Le format des données est ce qui définira le paramétrage des fonctionnalités de l'algorithme.

L'étape suivante est de créer et de dessiner les axes sur le `<canvas>`, il est important de déduire les mesures et l'échelle des axes en fonction des données chargées.

Il faut finalement placer les points sur le graphique, en ayant préalablement trié les points par ordre sur l'axe des abscisses afin de pouvoir les connecter dans l'ordre.

**p. 17 Solution n°4**

Les données qui étaient générées aléatoirement avec la fonction `generatePoints()` doivent être chargées en utilisant les fonctionnalités JSON de JavaScript.

La suite est de reprendre les fonctions de création des axes, pour faire la connexion entre les données chargées et les mesures qui seront déduites avec les données.

La dernière fonction à modifier est celle plaçant les points, la méthode `arc()` dessinant un point est désormais négligeable, et peut être remplacée ou couplée avec la méthode `lineTo()` pour ajouter des connexions entre chaque point qui se suivent.


**Exercice p. 17 Solution n°5****Question 1**

Lequel de ces attributs l'élément `<canvas>` ne possède-t-il pas ?

☐ `title`

☐ `width`

☒ `size`

 L'élément `<canvas>` possède l'attribut `title` qui est commun à tous les éléments HTML, et les attributs `width` et `height` qui lui sont spécifiques.

**Question 2**

La méthode `toDataURL()` de l'élément `<canvas>` permet d'obtenir le lien de l'image contenu.

☐ Vrai

☒ Faux

🔍 `toImgURL()` n'existe pas, la méthode `toDataURL()` est celle qui permet d'obtenir ce résultat. Le type de la donnée du lien est par défaut une image, mais elle est paramétrable.

### Question 3

La méthode `arc()` permet de former un arc avec les mesures de l'angle de début et l'angle de fin en radian. Quelles valeurs faut-il renseigner pour que ces 2 paramètres forment un cercle complet ?

- ☐ 0, Pi
- ☒ 0, 2\*Pi
- ☐ 0, 360
- ☐ - 180, 180

🔍 En trigonométrie, un cercle équivaut à 360 degrés ou 2 \* Pi radians, ainsi il faut que l'angle de début et l'angle de fin aient une différence de 2 \* Pi radians pour former un cercle complet.

### Question 4

La déclaration `Math.random() * (x_max - x_min)` génère une valeur aléatoire en `x_max` et `x_min`.

- ☐ Vrai
- ☒ Faux

🔍 Cette déclaration permet d'obtenir un nombre aléatoire dans un éventail de la même taille que ces limites, mais à partir de 0, il faut additionner `x_min` pour entrer dans les limites.

### Question 5

Dans la fonction `placePoints()`, quelle déclaration obtient la coordonnée finale en Y d'un point ?

- ☐ `50 + point.y / transform_y`
- ☐ `point.y / transform_y`
- ☒ `ctx.canvas.height - 50 - point.y / transform_y;`

🔍 L'axe Y étant inversé dans la zone de dessin du Canvas, il faut donc partir du bas du graphique, ce qui correspond à la valeur `ctx.canvas.height - 50`, et y soustraire la coordonnée du point proportionnelle au graphique, soit `point.y / transform_y`.