Laboratoire 7B : compléments sur les fonctions

Objectifs

- Se familiariser avec les diverses syntaxes avancées pour les fonctions (valeurs par défaut, surcharge, nombre de paramètres variable etc.)
- Se familiariser avec la notion de fonctions de haut niveau et leur utilisation
- Utiliser les fonctions de haut niveau pour gérer des réactions événementielles similaires ou encore pour retarder l'exécution du code
- Appliquer toutes ces connaissances dans le cadre d'exercices pratiques

Exercice 1 : Fonctions à définir _____ *2*

Exercice 2 : Fonctions comme objets de premier ordre _____ *3*

Exercice 3 : Mini-calculatrice _____ *5*

Exercice 4 : Redimensionnements _____ *9*

Exercice 5 : Le jeu de la vie _____ *12*

Exercice 1 : Fonctions à définir

Définissez et testez les fonctions demandées ci-dessous. Les crochets indiquent des arguments optionnels. (On propose également quelques exemples de tests).

1) `coteEnPourcents(cote [, max])` : donne la cote en pourcents (par défaut, max = 20)

```
coteEnPourcents(14)          → 70%
coteEnPourcents(15,30)       → 50%
```

2) `prixTVAC(prix [, tva])` : donne le prix TVA comprise (par défaut, tva = 21%)

```
prixTVAC(100)          → 121
prixTVAC(200,10)       → 220
prixTVAC(150,0)        → 150
```

3) `min (x1, x2, x3, …, xn)` : donne le minimum parmi toutes les valeurs citées
Modifiez la fonction min pour qu'elle puisse également être appelée avec un tableau de valeurs

```
min(2,3,1)             → 1
min(7,2,3,4)           → 2
min(15,20,21,42,53)    → 15
min([3,5,4])           → 3
```

4) `somme (x1, x2, x3, …, xn)` ou `somme (f, x1, x2, x3, …, xn)` : calcule la somme des valeurs données ou des résultats `f(x1)`, `f(x2)`, …, `f(xn)` si une fonction est fournie.

```
somme(1,2,3)                                    → 6
function carre(x) { return x*x; }
somme(carre,1,2,3)                              → 14
somme(function (x) { return 2*x; }, 1, 2, 3, 4)  → 20
somme(x => x*x, 1, 2, 3, 4)                     → 30
```

Exercice 2 : Fonctions comme objets de premier ordre

L'idée de considérer les fonctions comme des objets de premier ordre sert de base à toute une série de langages de programmation, des langages dits « fonctionnels ». Bien que Javascript ne soit pas un langage fonctionnel à proprement parler, il fournit de nombreuses situations où utiliser les fonctions comme objets de premier ordre simplifie grandement un code.

Étape 1 : quelques fonctions-tests

Définissez les fonctions suivantes dans un script Javascript directement dans la console ou au sein d'une page HTML vide, selon votre préférence :

- une fonction `plus2` qui reçoit une valeur et renvoie cette valeur augmentée de 2 ;
- une fonction `fois3` qui reçoit une valeur et renvoie cette valeur multipliée par 3 ;
- une fonction `cube` qui reçoit une valeur et renvoie cette valeur au cube.

Considérez le code suivant et tentez de comprendre à quoi correspond la fonction app2 ainsi définie.

```
function app2 (f, x) { return f(f(x)); }
```

Vérifiez si vous avez bien compris cette définition en déterminant (d'abord mentalement puis à l'aide de la console) la valeur des expressions suivantes. Cela va simplement effectuer la fonction 2x

```
app2(plus2,3)
app2(fois3,2)
app2(cube,2)
```

Étape 2 : afficheNFois

Définissez une fonction `afficheNFois(s, n)` qui écrit `n` fois la chaîne de caractères `s` dans la console (via `console.log`). On supposera que `n` est un entier positif.

Testez votre code sur deux ou trois exemples. N'oubliez pas de tester le cas où le second argument est 0 !

Note. Il y a au moins 2 manières différentes d'implémenter cette fonction (et les suivantes) : en utilisant une boucle ou en tant que fonction récursive. Vous êtes censés savoir comment coder les deux approches !

Modifiez votre code pour qu'on puisse également appeler la fonction en ne lui donnant qu'un seul argument (la chaîne de caractères). Dans ce cas-là, l'affichage devra être effectué 3 fois.

Testez votre nouveau code sur deux ou trois exemples. Ici encore, n'oubliez pas de tester le cas où on spécifie le second argument et que celui-ci est 0 !

Étape 3 : exécuteNFois

Basez-vous sur le code de la fonction précédente pour écrire une fonction `executeNFois(f, n)` qui exécute `n` fois la fonction/procédure `f` (on supposera que `f` ne prend aucun argument et qu'elle ne renvoie aucun résultat).

Entrez la définition de fonction

```
function message() { console.log("Ceci est un message !"); }
```

puis testez votre code en réalisant l'appel `executeNFois(message, 5)`.

Question : pourquoi, dans cet appel, n'écrit-on pas `message()` mais juste `message` ?

car il faut qu'on passe la définition de la fonction, ici ça l'exécuterait

Testez aussi votre code en utilisant une fonction anonyme ; par exemple :

```
executeNFois(function () { alert("Bouh !"); }, 3)
executeNFois(() => { console.log("Doh !"); }, 4)
```

Étape 4 : appn

Définissez une fonction `appn` qui prend comme arguments une fonction `f`, un nombre `n` et une valeur `x` et renvoie le résultat obtenu en appliquant `n` fois ($n \geq 0$) la fonction `f` à la valeur `x`.

Vérifiez votre code en effectuant quelques tests. Par exemple : `appn(plus2, 4, 3)` devrait valoir 11 car, si on applique 4 fois la fonction `plus2` à 3, on obtient

$$\Big(\big((3+2)+2\big)+2\Big)+2 = 11$$

et `appn(fois3, 3, 1)` devrait valoir 27.

Étape 5 : fonctions de haut niveau prédéfinies

Complétez les codes suivants pour qu'ils produisent le but attendu.

1) Complétez le code pour qu'il affiche la longueur de chacune des chaînes de caractères.

```
["un","deux","trois","quatre","cinq"].forEach(…)
```

2) Complétez le code pour qu'il retourne le tableau des éléments pairs.

```
[1,2,3,4,5,6,7,8,9,10].filter(…)
```

Exercice 3 : Mini-calculatrice

Le but de cet exercice est de réaliser une mini-calculatrice en Javascript.

Étape 1 : le document de base

Considérez tout d'abord le fichier HTML suivant. Examinez son contenu.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8"/>
  </head>
  <body>
  <p>Valeur actuelle : <span id="spanVal"></span></p>
    <p>Opérations :
      <button id="bOppose">+/-</button>
      <button id="bCarre">carré</button>
      <button id="bFact">factorielle</button>
    </p>
    <p>Ajuster :
      <button id="bDecremente10">-10</button>
      <button id="bDecremente1">-1</button>
      <button id="bRAZ">0</button>
      <button id="bIncremente1">+1</button>
      <button id="bIncremente10">+10</button>
    </p>
    <p>Multiplier :
      <button id="bMultiplie">×2</button>
      <button id="bDeux">2</button>
      <button id="bCinq">5</button>
      <button id="bDix">10</button>
    </p>
  </body>
</html>
```

Le premier paragraphe permet d'afficher la valeur sur laquelle les opérations s'effectueront. Viennent ensuite trois autres paragraphes chaque fois avec plusieurs boutons correspondant à diverses opérations :

- Dans le premier paragraphe, les trois boutons correspondent aux opérations :
  - prendre l'opposé (changer de signe) : 3 devient -3, -7 devient 7 ;
  - remplacer la valeur actuelle par son carré ;
  - remplacer la valeur actuelle par sa factorielle (0! = 1! = 1, 2! = 2, 3! = 3×2×1, …) ou ne rien faire si la valeur actuelle est négative.
- Dans le second paragraphe, on trouve trois boutons permettant d'ajuster la valeur, soit en la décrémentant / en l'incrémentant (de 1 ou de 10), soit en la remettant directement à zéro.

- Finalement, dans le troisième paragraphe se trouve un bouton permettant de multiplier la valeur par un certain nombre. L'effet du premier bouton (la multiplication) pourra être modifié en cliquant sur un des trois boutons qui suivent, afin de multiplier par 2, par 5 ou par 10.

Dans cet exercice, on vous propose d'implémenter chacune des trois lignes de boutons de manière différente.

Étape 2 : mise en place

Tout d'abord, commencez par établir un espace de travail. Ajoutez dans le fichier HTML un lien vers un fichier Javascript (ou travaillez directement avec du code Javascript placé dans la balise `<head>` si vous préférez).

Déclarez-y une variable qui servira à stocker la valeur actuelle, ainsi qu'une fonction `majValeur` permettant de mettre à jour l'affichage de cette valeur (en modifiant le contenu de l'élément `spanVal`). Cette fonction sera appelée par la plupart des actions associées aux boutons.

Arrangez-vous pour que la valeur soit initialisée à 2. Ça peut sembler être un choix étonnant (pourquoi pas 0 ?), mais cela nous sera utile pour pouvoir tester les premiers boutons. Par la suite, vous pourrez modifier la valeur par défaut pour que la calculatrice affiche « 0 » au lancement.

Pour bien faire, il faudrait que la fonction de mise à jour de l'affichage soit appelée une première fois dès la fin du chargement de la page. Ajoutez au script le code nécessaire pour cela en utilisant la ligne suivante et en définissant la fonction `init` de manière adéquate.

```
window.onload = init;
```

Étape 3 : les trois premiers boutons

Définissez trois fonctions permettant d'effectuer les calculs nécessaires, à savoir :

- `calcOpposé()` : remplace la valeur par son opposé ;
- `calcCarré()` : remplace la valeur par son carré ;
- `calcFactorielle()` : remplace la valeur par sa factorielle (définissez une fonction annexe `factorielle(x)` soit de manière récursive, soit en utilisant une boucle).

Liez ces fonctions aux boutons en suivant les consignes suivantes :

- Pour l'opposé et le carré, modifiez directement le code HTML.
- Pour la factorielle, arrangez-vous pour associer l'action au bouton après le chargement de la page (via du code Javascript donc).

Étape 4 : le second groupe de boutons

Occupez-vous tout d'abord du bouton remettant la valeur à zéro. Associez-lui son action via la fonction `init`, comme ci-dessus.

Les quatre boutons suivants ont un effet similaire : +1, +10, -1, -10. Dans tous les cas, il s'agit d'ajouter un certain nombre (qui peut être négatif) à la valeur actuelle. Ce serait commode de n'écrire le code qu'une seule fois, de sorte qu'on puisse ajouter à `init` les lignes suivantes.

```
document.getElementById("bDecremente10").onclick = calcAjoute(-10);
document.getElementById("bDecremente1").onclick = calcAjoute(-1);
document.getElementById("bIncremente1").onclick = calcAjoute(1);
document.getElementById("bIncremente10").onclick = calcAjoute(10);
```

Pour que cela soit possible, il faudrait que chacune des expressions à droite du signe `=` soit une fonction. Autrement dit, il faut que `calcAjoute(-10)`, `calcAjoute(-1)`, `calcAjoute(1)` et `calcAjoute(10)` soient des fonctions. Qu'est-ce que cela implique à propos de `calcAjoute` et de son type de retour ?

Il faut que `calcAjoute` soit une fonction qui reçoit un nombre (-10, -1, 1 ou 10) et qui renvoie… une fonction ! Sur base du nombre qu'elle reçoit, `calcAjoute` construit une fonction et la renvoie.

Définissez la fonction `calcAjoute`. Il y a plusieurs manières de le faire (pensez notamment aux diverses manières d'écrire une fonction en Javascript : notation standard, notation littérale…).

Ce processus consistant à définir une fonction qui renvoie l'action à associer à un élément HTML est utilisé à nouveau dans l'étape suivante ; assurez-vous donc de bien le comprendre ! Il s'avère tout particulièrement pratique lorsqu'il faut traiter une série d'éléments HTML qui ont des effets similaires.

Testez la calculatrice.

Étape 5 : le troisième groupe de boutons

Le dernier groupe de boutons… parmi ceux-ci, un seul réalise véritablement une opération : il s'agit du premier. Cette opération est ×2, ×5 ou ×10. Initialement, lors du lancement de la calculette, l'opération est ×2 mais, si l'utilisateur clique sur l'un des trois boutons qui suivent, l'opération change (pour devenir respectivement ×2, ×5 ou ×10).

Tout d'abord, comme dans l'étape précédente, définissez une fonction `calcMultiplie(x)` qui renvoie une fonction procédant au calcul et rafraîchissant l'affichage. Dans la suite, on utilisera `calcMultiplie(2)`, `calcMultiplie(5)` et `calcMultiplie(10)`.

Assurez-vous qu'initialement, on associe au bouton de multiplication l'opération `calcMultiplie(2)` : modifiez la fonction `init` en conséquence.

Que devra réaliser un clic sur le bouton « ×5 » ? Principalement deux choses : (1) modifier le texte du bouton de multiplication et (2) modifier l'action du bouton de multiplication.

Les boutons « x2 » et « x10 » auront des effets similaires.

Comme l'effet sera quasiment le même pour les trois boutons (×2, ×5 et ×10), au lieu de répéter du code similaire, il vaudrait mieux ne l'écrire qu'une seule fois et utiliser un paramètre (un argument) pour ce qui change (à savoir la valeur 2, 5 ou 10).

Définissez donc une fonction `effetBoutonFois` qui recevra comme argument 2, 5 ou 10 et qui renverra l'action (c'est-à-dire une fonction) associée au bouton en question. Associez ces actions aux boutons adéquats dans la fonction `init`.

Exercice 4 : Redimensionnements

Le document HTML de cet exercice affiche trois blocs colorés de tailles différentes (l'un est nettement plus grand que les deux autres). À eux trois, ces blocs colorés doivent se partager l'emplacement déterminé par un cadre. D'un simple clic, l'utilisateur devra pouvoir choisir lequel des trois blocs colorés va occuper la place la plus importante.

C'est un mécanisme d'affichage dynamique assez souvent utilisé dans les sites web (parfois appelé « accordéon » à cause des différentes parties qui se plient et se déplient). Certains outils, comme jQuery, permettent de réaliser des présentations de très grande qualité mais, ici, pour l'exercice, on va se contenter d'utiliser des fonctionnalités Javascript.

Étape 1 : le document de départ

Voici le fichier HTML de départ.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8"/>
    <style>
      #cadre { border: 1px solid black; margin: auto;
               width: 600px; height: 220px; }
      #cadre div { height: 200px; margin: 10px; float: left; }
      #bloc0 { background-color: blue; width: 500px; }
      #bloc1 { background-color: red; width: 20px; }
      #bloc2 { background-color: green; width: 20px; }
    </style>
  </head>
  <body>
    <div id="cadre">
      <div id="bloc0" onclick="clic(0);"></div>
      <div id="bloc1" onclick="clic(1);"></div>
      <div id="bloc2" onclick="clic(2);"></div>
    </div>
  </body>
</html>
```

Le div `cadre` délimite l'espace disponible : 600px × 220px. À l'intérieur de celui-ci, on trouve trois div identifiés par `bloc0`, `bloc1` et `bloc2` faisant tous 200px de hauteur (+ 10px de marge en haut et en bas pour arriver au 220px du cadre) et ayant des couleurs différentes.

Horizontalement, les 600px sont divisés ainsi (de gauche à droite) :

- 10 px de marge ;
- 500 px pour le `bloc0` (bleu) ;
- 20 px de marge entre deux blocs ;
- 20 px pour le `bloc1` (rouge) ;
- 20 px de marge entre deux blocs ;

- 20 px pour le `bloc2` (vert) ;
- 10 px de marge.

Les « petits » blocs auront donc une largeur de 20px alors que le seul et unique grand s'étendra sur une largeur de 500px.

Étape 2 : un clic pour redimensionner les blocs

Pour mémoriser le numéro du grand bloc, ajoutez une variable globale `grandActuel` dans le script Javascript (elle sera initialisée à 0). Notez que, dans le fichier HTML, en cas de clic sur l'un des blocs, on appelle la fonction `clic` avec, comme argument, le numéro du bloc sur lequel on a cliqué.

Écrivez cette fonction `clic` sachant que

- si le numéro de bloc donné est déjà celui du `grandActuel`, elle ne doit rien faire ;
- dans le cas contraire, elle doit mettre la largeur de l'ancien grand à 20px, celle du nouveau grand à 500px et mettre à jour la variable `grandActuel`.

Note. Pour accéder à un cadre intérieur dont vous connaissez le numéro, vous pouvez passer par le grand cadre et utiliser `children`.

Note (2). Pour modifier la largeur d'un élément, comme il s'agit d'un composant de style, il faut utiliser `elem.style.width = "20px"` (n'oubliez pas les unités). Pour un code plus clair, définissez une fonction `modWidth` qui prend comme arguments le numéro d'un cadre (0, 1 ou 2) et la largeur qu'il faut lui donner et effectue la modification.

Étape 3 : redimensionnement progressif

Améliorons le côté esthétique : plutôt que de modifier les largeurs de 500px à 20px (et inversement) en une seule fois, pourquoi ne pas faire une boucle qui les modifie peu à peu. Visuellement, cela devrait donner une progression plus plaisante au lieu d'un changement brusque.

La progression pourrait s'effectuer de 10px en 10px.

Ainsi, initialement, l'ancien grand mesure 500px et le futur grand, 20px. Après une étape, le premier mesurerait 490px et le second, 30px. Après une seconde étape, on arriverait à 480px pour le premier et 40px pour le second. Et ainsi de suite…

Modifiez la fonction `clic` en remplaçant les modifications de largeur par une boucle progressive.

Étape 4 : ralentir le redimensionnement

Malheureusement, la progression reste trop rapide pour que l'œil humain puisse véritablement l'observer. Comment la ralentir ? En donnant des ordres à effectuer dans le futur, ce qui implique d'utiliser `setTimeout`.

La fonction `setTimeout` (en fait, il s'agit d'une méthode de l'objet global `window`) permet de demander l'exécution d'une fonction dans un certain nombre de millisecondes dans le

futur. Ici, il s'agira de demander l'exécution dans le futur de deux modifications de largeur (celle portant sur l'ancien grand et celle portant sur le nouveau grand).

Révisez le code de la fonction `clic` pour qu'elle planifie les appels à `modWidth` (vous pouvez faire une étape — c'est-à-dire ajouter/retirer 10px de largeur — toutes les 10 millisecondes par exemple).

Exercice 5 : Le jeu de la vie

Le « jeu de la vie » est un petit jeu censé simuler la manière dont la vie se propage sur une grille quadrillée où chaque cellule peut être soit vivante soit morte. À chaque étape, l'état d'une cellule peut changer, c'est-à-dire passer de vivante à morte (la cellule meurt) ou passer de morte à vivante (la cellule naît).

Les changements d'état se font selon des règles très précises qui se basent sur le nombre de cellules vivantes parmi les 8 cellules voisines (en ligne droite et en diagonale) :

- Une cellule vivante qui a 2 ou 3 voisines vivantes reste vivante… sans ça, elle meurt.
- Une cellule morte qui a exactement 3 voisines vivantes naît. Sans ça, elle reste morte.

Selon la configuration initiale des cellules vivantes, l'évolution du jeu de la vie peut donner lieu à diverses figures aux propriétés étonnantes (plus d'informations sur Google).

Étape 1 : le document de départ

Voici le fichier HTML de départ.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8"/>
    <style>
      td { width: 20px; height: 20px;
           border: 1px solid black; background-color: white; }
      td.vivant { background-color : blue; }
    </style>
  </head>
  <body>
    <p>
      <button id="bAvancer">Avancer</button>
      <button id="bContinu">Continu</button>
      <button id="bStop" disabled="disabled">Stop</button>
    </p>
    <table id="plateau"></table>
  </body>
</html>
```

La page affiche trois boutons intitulé Avancer, Continu et Stop et définit une table HTML (vide, pour l'instant) d'identificateur « plateau ». Les styles définis en début de fichier indiquent que les cases du tableau seront des carrés encadrés de fond blanc et que celles qui possèdent la classe « vivant », elles, auront un fond bleu.

Étape 2 : création du plateau de jeu

Dans un premier temps, ajouter du code Javascript pour qu'au chargement de la page, on demande le nombre de lignes et de colonnes de la grille de jeu. Ne vous préoccupez pas trop des éventuels dépassements de taille (on supposera que l'utilisateur rentre des valeurs acceptables).

Prévoyez également une fonction pour « remplir » le tableau avec le nombre de lignes et de colonnes demandées. Cette fonction pourrait utiliser `innerHTML` mais, pour l'exercice, contraignez-vous à employer plutôt `document.createElement` pour chaque ligne (`tr`) et chaque case (`td`) ; cela se révélera plus pratique pour la suite.

Étape 3 : représentation en mémoire du plateau de jeu

On va souvent devoir intervenir sur les cases de la grille. Pour retrouver l'élément HTML qui correspond à une case de coordonnées i, j données, on pourrait passer à chaque fois par la recherche de la table HTML puis utiliser `children[i]` pour trouver la bonne ligne et encore `children[j]` pour trouver la bonne cellule.

On peut aussi décider de stocker dans une matrice / un tableau à 2 dimensions (des pointeurs vers) les cellules. Le meilleur moment pour le faire est lorsque les éléments HTML sont créés, c'est-à-dire dans la fonction écrite à l'étape précédente.

Ajoutez à votre code une variable globale `table` qui représentera le tableau de jeu. Garnissez-la au fur et à mesure de la construction de la table de manière à ce que `table[i][j]` soit l'élément HTML correspondant à la cellule située sur la ligne i (= 0, 1, 2…) et la colonne j (= 0, 1, 2…).

Note. Revoyez les slides concernant les tableaux : il faut initialiser ceux-ci sous la forme de tableaux vides avant de pouvoir y ajouter des éléments. Pour les matrices (= tableaux de tableaux), il faut non seulement initialiser la matrice mais également chacune de ses lignes !

Étape 4 : configuration de départ

L'utilisateur pourra indiquer la configuration de départ en cliquant tout simplement sur les cases à rendre vivantes. S'il clique par erreur sur une case, il pourra cliquer une seconde fois pour la remettre à son état initial. Il faut donc associer à chacune des cases une action pour l'événement `click` qui va se charger de modifier son état (vivante/morte).

Définissez une fonction `clic` qui sera l'action déclenchée lors d'un clic sur une des cases du tableau. Liez l'action à chacune des cases du tableau.

Note. Rendre une cellule vivante revient à lui ajouter la classe CSS `vivant` ; rendre une cellule morte revient à lui retirer la classe CSS `vivant`. La définition de la fonction `clic` peut donc tenir en une ligne !

Étape 5 : une étape dans le jeu de la vie

Cette étape vise à implémenter l'action associée au bouton « Avancer ». Celui doit faire avancer l'état global de la grille d'une étape. Cela revient à modifier l'état (a) des cellules vivantes qui ont moins de 2 ou plus de 3 voisins et (b) des cellules mortes qui ont exactement 3 voisins.

Cette partie relève plutôt de l'algorithmique générale… Vous avez donc le champ libre, mais pensez à écrire des fonctions annexes qui pourront faciliter l'écriture et la relecture de votre code. Si l'énoncé ne vous semble pas clair, n'hésitez pas à consulter la page « jeu de la vie » sur Wikipédia.

Note. Pour savoir si une cellule doit changer d'état, on se base sur l'état de ses voisines *au début* de cette étape. Ainsi, si lors d'une étape, une cellule a une voisine qui meurt, cette voisine comptera comme « vivante » lorsqu'il faudra déterminer le sort à réserver à la cellule.

Note (2). Les cases situées au bord de la grille ont évidemment moins de voisines que les autres. En fait, tout se passe pour elles comme si toutes les cases « voisines » en-dehors de la grille était mortes.

Étape 6 : la vie en continu

Le plus gros du boulot a été fait, mais il reste deux boutons : Continu et Stop. En cliquant sur le premier, l'utilisateur pourra déclencher l'avancement « continu » du jeu de la vie, avec une mise à jour de la grille toutes les demi-secondes. En cliquant sur le second, il pourra stopper l'avancement en continu.

C'est une bonne occasion pour utiliser `setInterval`, la méthode permettant de demander l'exécution d'une fonction à intervalles réguliers.

Pour améliorer l'interface, on va également griser les boutons qui ne sont pas utilisables. Ainsi, au départ, le bouton « Stop » est grisé (`disabled`). Lorsque l'utilisateur aura cliqué sur Continu, c'est ce dernier qui deviendra grisé et « Stop » sera accessible. Quand l'utilisateur appuiera alors sur « Stop », les deux boutons reviendront à leur état de départ.

Note. Si `b` désigne un élément HTML, on peut le rendre inactif avec l'instruction `b.disabled = true` puis le rendre à nouveau actif avec `b.disabled` à false.

Pour en apprendre plus sur le jeu de la vie… une vidéo Youtube repérée par un ancien étudiant : https://www.youtube.com/watch?v=S-W0NX97DB0 – bon visionnage !