Gestion des bugs : stratégies générales

Table des matières

I. Contexte

Durée : 1 h

Environnement de travail : Chrome, Firefox, VisualStudio Code

Pré-requis : Aucun

Contexte

Les bugs sont indissociables du développement. Même les meilleurs développeurs en introduisent, et leurs résolutions font partie du quotidien.

C'est pourquoi il est important de connaître les outils et méthodes qui vont faciliter le débogage.

II. Comprendre le code

Objectif

- Apprendre à analyser du code

Mise en situation

Parce que "l'erreur est humaine", il arrive au développeur d'écrire un code qui va produire une erreur : il introduit un **bug** dans son code. Il est donc très important de savoir identifier la cause de la défaillance pour pouvoir la corriger au plus vite.

Les types de bugs

Voici les trois principaux types de bugs :

- Les bugs de **syntaxe** :

Dus à une faute de frappe, un caractère en trop ou manquant. Ces bugs seront généralement signalés par l'interpréteur JavaScript.

```
var myText = 'Mon texte; //Il manque ici une apostrophe en fin de texte
/*
L'interpréteur pourra retourner un message comme :
"Unclosed string. Missing semicolon. Line 1."
ou simplement
"SyntaxError"
*/
```

- Les bugs de **logique** :

Dus à une erreur dans le raisonnement du développeur ou dans l'écriture du script.

```
//Ici une fonction censée additionner ses 2 paramètres et renvoyer la valeur de la somme
function Sum(first, second) {
    var result = first + second;
    return first; //Le résultat renvoyé n'est pas celui attendu.
}
```

- Les bugs d'**architecture** :

Dus à une erreur dans la conception du programme, une référence manquante ou tout ce qui touche à la structure du code.

```
//Ici, on essaie d'utiliser la méthode myAwesomeFunction, présente dans un fichier distant, sans importer le module.

myAwesomeFunction('David', 'Marty', 'USAP');
```

```
//Version correcte :
import { myAwesomeFunction } from '/modules/fichier.js';

myAwesomeFunction('David', 'Marty', 'USAP');
```

Comprendre le code

Avant de déboguer un code, il faut le comprendre. Quel est le but ? Que font telles ou telles fonctions ? Quelles sont les étapes pour arriver au résultat ?

Déterminer le but du code revient à déterminer le résultat attendu.

Ici, le résultat attendu est l'affichage d'une alerte :

```
function showAlert(alert) {
   alert(alert);
}

const alert = 'Hello World';
showAlert(alert);
```

Toutefois, tous les codes ne sont pas si simples et il faudra souvent passer à travers une série de fonctions pour comprendre le résultat attendu. En déduisant ce résultat pour chaque fonction et en connaissant le but de celles-ci, on pourra :

- Comparer le résultat attendu avec le résultat réellement retourné par la fonction
- Réorganiser le code pour le simplifier (factoriser une fonction complexe en plusieurs fonctions simples, supprimer des fonctions superflues, etc.)

Reprenons le cas de notre script affichant une alerte, en complexifiant un peu.

```
<!doctype html>
<html lang="fr">
  <head>
    <meta charset="utf-8">
    <title>Exemple</title>

  </head>
  <body>
    <form>
        <input type="text" name="alerte"></input>
        <button type="button" onclick="getAlert(alerte.value)">Alerte</button>
    </form>
  </body>

  <script>
    function showAlert(alert) {
      alert(alert);
    }

    function getAlert(alerte) {
      showAlert(alert)
    }

  </script>
</html>

```

Ici, nous avons ajouté un input de type texte et un bouton. Lorsque l'on clique sur le bouton, on appelle la fonction `getAlert()` qui prend en paramètre la valeur de l'input. Cette fonction appelle elle-même une autre fonction, `showAlert()`, dont le rôle est d'afficher l'alerte.

Dans ce code, nous avons donc deux fonctions dont nous pouvons déduire les rôles selon leur nom :

- `getAlert('alert')` : récupère l'alerte,
- `showAlert('alert')` : affiche l'alerte.

Ce code, aussi simple soit-il et bien qu'il soit correct, dispose tout de même d'une incohérence **logique**. On remarque que la fonction `getAlert()` appelle directement la fonction `showAlert()`.

Cela revient à dire que la fonction `getAlert()` a pour rôle à la fois de récupérer l'alerte et de la montrer. Pour rétablir la cohérence, il faut réfléchir à l'utilité et au nommage des fonctions.

Il serait plus logique de mixer les 2 fonctions en 1 et que notre fonction `getAlert()` s'appelle `showAlert()`.

```
<!doctype html>
<html lang="fr">
  <head>
    <meta charset="utf-8">
    <title>Exemple</title>

  </head>
  <body>
    <form>
        <input type="text" name="alerte"></input>
        <button type="button" onclick="showAlert(alerte.value)">Alerte</button>
    </form>
  </body>

  <script>
    function showAlert(alert) {
      alert(alert);
    }
  </script>
</html>

```

Nous avons enlevé de la complexité et ajouté de la clarté à notre code.

Toutefois, ce n'est pas si simple et, la plupart du temps, déboguer un code est bien plus difficile. Il existe des méthodes de débogage plus poussées, que nous verrons dans les chapitres suivants.

Syntaxe **À retenir**

Pour déboguer un code, il faut :

- Comprendre son but,
- Comprendre les fonctions qui le composent,
- Rétablir la logique en renommant les variables et les fonctions correctement, par exemple,
- Réduire sa complexité si nécessaire.

III. Exercice : Appliquez la notion

Question 1

Vous devez travailler sur un algorithme qui retourne un tirage aléatoire du loto. Le résultat ne correspond pas à ce qui est attendu et vous devez réparer ce code.

`winNumbers` retourne la liste des nombres gagnants.

`complementaryNumbers` retourne la liste des numéros complémentaires.

```
const listeNumber = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24,
25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49]

const winNumbers = [];
const complementaryNumbers = []

function randomInInterval(min, max){
  return Math.floor(Math.random() * (max - min + 1)) + min;
}

function addToTable(table, tableRef) {
  const index = randomInInterval(0, tableRef.length - 1)
  table.push(tableRef[index]);
  tableRef.splice(1, index);
}

function tirage(size, table, tableRef) {
  while (table.length < size) {
    addToTable(table, tableRef)
  }
}

tirage(5, winNumbers, listeNumber);
tirage(2, complementaryNumbers, listeNumber);

console.log(winNumbers, complementaryNumbers)
```

Testez plusieurs fois ce code et regardez le résultat en console. Que remarquez-vous ? Quel est le problème du résultat de ce code ?

Question 2

À l'aide de la documentation et de vos connaissances, analysez chaque fonction et placez des commentaires expliquant le rôle de chacune d'elles.

Question 3

À présent, nous savons ce que chaque fonction fait et nous allons donc simplifier le code pour pouvoir le tester.

La fonction `addToTable()` est appelée dans une boucle. On peut très bien tester cette fonction hors de la boucle et avec un jeu de données plus simple.

Testez plusieurs fois la fonction avec les données de test ci-dessous et regardez le résultat en console. Qu'en déduisez-vous ?

```
const testRef = [1, 2, 3];
const testToGet = [];
```

- `testRef` représente `listeNumber`
- `testToGet` représente `winNumbers`

Question 4

Maintenant que vous avez ciblé le problème en simplifiant le code, rendez-vous dans la documentation de la méthode `splice()`, analysez le problème et corrigez la fonction.

IV. Déboguer dans le navigateur

Objectif

- Connaître les outils à disposition du développeur par le navigateur Internet

Mise en situation

Les navigateurs modernes mettent à disposition tout un panel d'outils qui vont faciliter le débogage JavaScript, entre autres.

Fondamental **Outils de développement Firefox et Chrome**

Tous les navigateurs modernes possèdent des outils de développement, ou *DevTools*. À ce jour, les plus évolués sont ceux de Firefox et de Chrome.

Pour y accéder, il suffit d'ouvrir le navigateur et de taper sur la touche **F12** du clavier, ou **Ctrl** + **Shift** + **i** sous Windows et Linux, **Cmd** + **Opt** + **i** sous macOS.

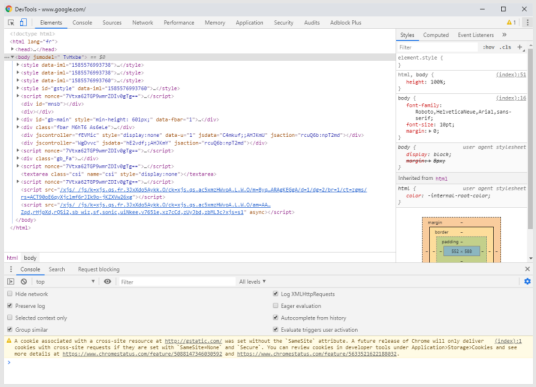
La fenêtre **DevTools** apparaît. Elle pourra être attachée ou détachée de la fenêtre du navigateur en cours.

Plusieurs onglets sont présents et en rapport avec la page web actuellement affichée dans le navigateur.

Méthode **L'onglet elements**

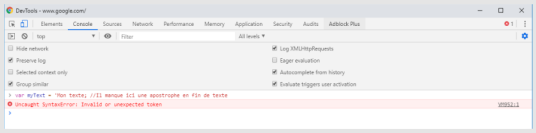
Ici, l'onglet actif est **Elements**. Cet onglet permet d'afficher et de modifier le code HTML et CSS de la page en cours, pour un rendu immédiat.

Ctrl + **Maj** + **C** (ou **Cmd** + **Opt** + **C** sous macOS) permettra de sélectionner des éléments de la page web dans la fenêtre principale et de centrer le code HTML concerné.



Méthode **L'onglet console**

La **console** est présente sous forme d'un onglet à part, mais aussi dans chacun des onglets de DevTools (appuyez sur **ESC** dans n'importe quel onglet afficher/masquer).



La console a plusieurs utilités :

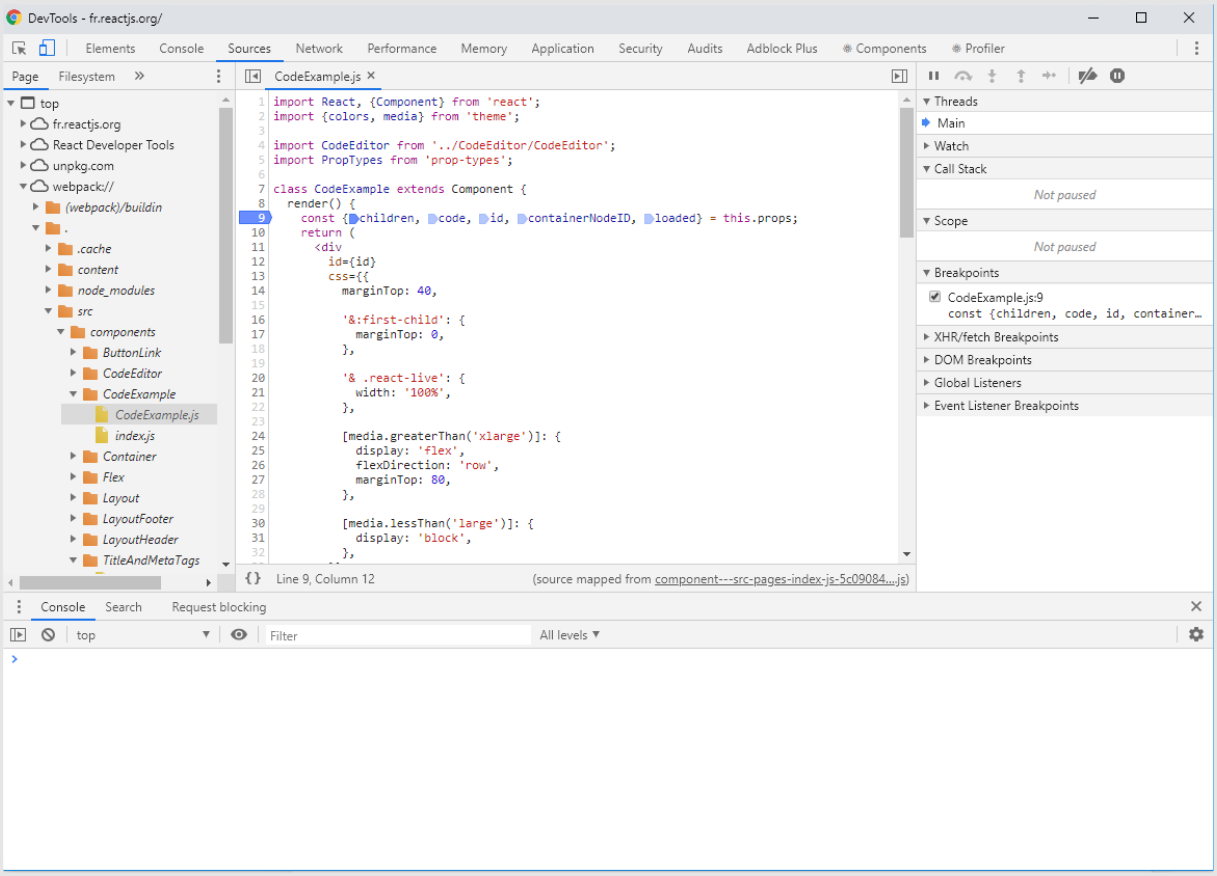
- C'est ici que vont s'afficher tous les messages d'erreurs et d'avertissements levés lors de l'exécution du code. C'est également ici que s'afficheront tous les retours de `console.log()`.
- Il est possible d'exécuter du code JavaScript directement dans la console, en appuyant sur **ENTER** après avoir écrit votre script. La console exécutera le code et pourra retourner un message d'information, d'erreur, un warning, etc.

Méthode **L'onglet sources**

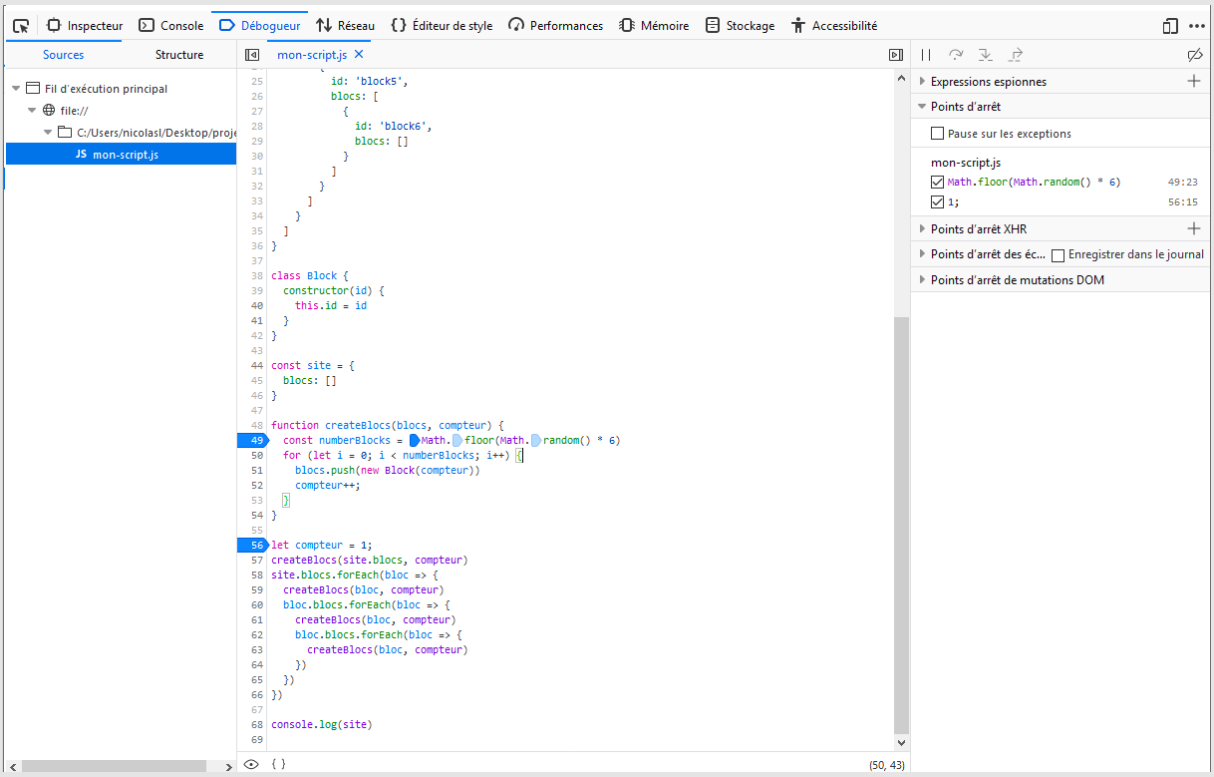
L'onglet **Sources** fait sans doute partie des onglets les plus utiles aux développeurs. Il permet d'explorer le code source de l'application web.

On pourra par exemple naviguer dans l'arborescence de l'application et explorer tous les fichiers qui la composent.

Console Google chrome :



Console Mozilla Firefox :



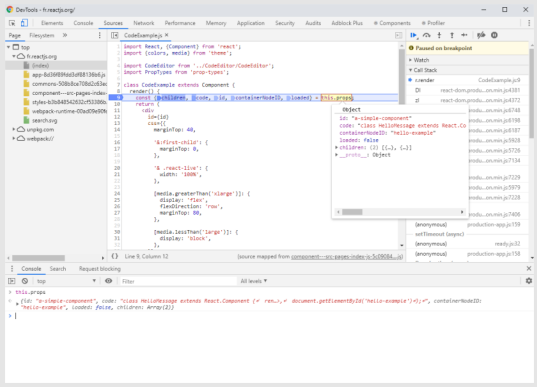
Un des très grands services que rend l'onglet **Sources** est la possibilité d'insérer des points d'arrêt et de faire du "pas-à-pas" sur le code de l'application.

Pour ajouter un point d'arrêt, il suffit de cliquer sur le numéro de la ligne à laquelle on veut s'arrêter, et de relancer l'exécution du code.

La ligne en cours s'affiche en surbrillance. Il est désormais possible d'évaluer les objets JavaScript, soit par un survol du curseur, soit en tapant le nom de l'objet dans la console.

Il est également possible d'instancier des variables, propriétés, ou autres via la console.

Pour avancer "pas-à-pas" dans le code, on utilisera les flèches en haut à droite de la fenêtre.



Complément **Le mot-clé debugger**

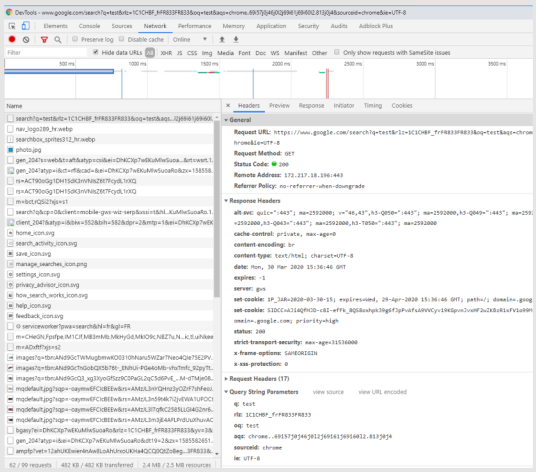
En JavaScript, il existe le mot clé **`debugger`**. L'avantage de ce mot-clé est que, si DevTools est ouvert, l'exécuteur JavaScript fera une pause lorsqu'il rencontrera le mot-clé `debugger` dans le code.

```
function Sum(first, second) {
    var result = first + second;
    debugger      //l'execution du code fera un arrêt ici
    return result;
```

```
5 }
```

Méthode **L'onglet network**

L'onglet **Network** va nous permettre d'intercepter l'ensemble des requêtes HTTP envoyées depuis l'application web.



Pour chacune de ces requêtes, nous obtiendrons des informations utiles, comme :

- L'URL de destination
- La méthode HTTP
- Le statut de la requête
- Les headers de la requête et de la réponse
- La réponse
- Le délai de traitement de la requête
- etc.

Ces requêtes pourront être filtrées selon leurs types respectifs. On pourra choisir, par exemple, de n'afficher que les requêtes `XHR` ou `Fetch`.

Complément **Extensions**

Il existe une multitude d'extensions navigateur qui permettent de faciliter le développement dans des cas bien précis. Elles seront disponibles sur les MarketPlaces respectives de chaque navigateur.

On peut citer, par exemple, *React Developer Tools* pour le développement React.

Syntaxe **À retenir**

- Tous les navigateurs web modernes proposent des outils d'aide au développement d'application web. Pour Chrome et Firefox, des outils indispensables comme l'inspecteur, la console web, le débogueur JavaScript, et l'intercepteur Network sont accessibles via la touche **F12**.

Complément

Chrome DevTools [1]

Firefox DevTools[2]

V. Exercice : Appliquez la notion

Question

Le code suivant est censé retourner le nombre de notes égales à 15. Lorsque l'on exécute ce code, le nombre affiché en console est 5. Or, on sait que ce nombre devrait être 1. Ce code contient donc une erreur.

Copiez-collez ce code dans un fichier HTML et exécutez-le via votre navigateur.

Utilisez les notions apprises dans le cours pour placer des points d'arrêts, et trouvez l'erreur en faisant du pas-à-pas.

[1] https://developers.google.com/web/tools/chrome-devtools
[2] https://developer.mozilla.org/fr/docs/Outils

```html
<!doctype html>
<html lang="fr">
<head>
  <meta charset="utf-8">
  <title></title>
</head>
<script>

const notes = [25, 18, 16, 15, 12]
const noteToCompare = 15;

let notesAccepted = 0;

for (let i = 0; i < notes.length; i++) {
  if (notes[i] = noteToCompare) {
    notesAccepted += 1
  }
}

console.log(notesAccepted) //5

</script>

</html>
```

VI. Résolution d'un bug : les étapes

Objectif

- Apprendre une méthodologie d'analyse et de résolution des bugs

Mise en situation

Le débogage est un aspect particulier et très exigeant du métier de développeur. Sans les bons outils et une bonne méthodologie, cela peut vite devenir un casse-tête.

Ce cours apportera les principales étapes pour un débogage efficace.

Reproduire l'erreur

Lorsqu'un bug est remonté, il est primordial d'arriver à reproduire l'erreur. Sinon, comment s'assurer de la bonne résolution du bug ? Il est donc important d'avoir un scénario de reproduction complet avec données.

Malgré tout, il faut aller au plus simple lors de la rédaction d'un scénario, ne garder que les étapes indispensables à la reproduction. Ce scénario sera joué de nombreuses fois au cours du débogage, l'idéal serait de pouvoir déclencher le bug d'un clic, comme avec un test unitaire.

L'apparition d'un bug est parfois liée au contexte d'exécution du code. En effet, les environnements de production, d'intégration ou de développement peuvent avoir des différences d'infrastructure, de logiciels, de données, qui vont jouer sur l'apparition de bugs. Il est très fortement recommandé d'avoir un environnement de développement qui se rapproche le plus possible de l'environnement de production.

De plus, faire du *debug* en local est bien plus simple que sur un environnement d'intégration ou en production. Plus le bug est détecté tôt, moins il en coûtera en correction.

Si le bug n'est pas reproductible, il faut trouver pourquoi :

- A-t-il été corrigé par une modification du programme ? Il faut trouver laquelle.
- Est-il lié aux données ? Il faut importer ou créer des données en local (*mock*).
- Est-ce lié à l'environnement ? S'il n'est pas possible de reproduire la configuration en local, il faudra déboguer sur cet environnement, à l'aide de logs par exemple.

Une fois le scénario défini, il est important de l'étendre à d'autres données pour identifier les contours du bug.

Exemple

Erreur constatée : Quand on clique sur le lien "Utilisateur 1", sa page de profil ne s'ouvre pas.

Scénario principal de reproduction : Depuis la page X, on clique sur le lien "Utilisateur 1".

On va créer un scénario avec un clic sur le lien "Utilisateur 2". Si le bug apparaît, c'est probablement un problème avec l'ouverture de la page, sinon c'est un probablement un problème avec les données de l'utilisateur 1.

Produire des hypothèses

Maintenant que nous arrivons à reproduire l'erreur, il est temps de formuler des hypothèses quant à sa source. Un bug peut apparaître dans trois conditions :

- C'est la première fois que l'on teste le comportement en erreur,
- L'environnement du projet a changé (données, librairies externes...),
- L'erreur a été introduite avec un nouveau code ajouté au projet.

Pour pouvoir formuler des hypothèses correctes, il faut maîtriser toutes les variables qui vont être utilisées dans la partie du projet incriminée. Pour cela, il est souvent judicieux de créer des *Mocks* de données.

Conseil

Il ne faut pas sous-estimer l'importance de l'expérience dans le diagnostic, une erreur similaire est peut-être déjà arrivée à un collègue. N'hésitez pas à échanger.

Effectuer un diagnostic

La stratégie la plus employée pour établir un diagnostic est le *Back-tracking*. Il s'agit de partir de la ligne de code d'où le bug a été constaté, et de remonter les instructions jusqu'à trouver la source du bug.

Les étapes sont simples :

1. Ajouter un point d'arrêt au niveau de la ligne de code en erreur,
2. Exécuter le code, selon le scénario de reproduction du bug,
3. Arrivé au point d'arrêt, analyser les valeurs des variables et définir laquelle provoque l'erreur,
4. Remonter jusqu'à l'instanciation de cette variable avec la valeur en erreur et y ajouter un point d'arrêt. Supprimer les points d'arrêt précédemment ajoutés, pour plus de clarté,
5. Répéter les étapes 2 à 4, jusqu'à l'origine du bug.

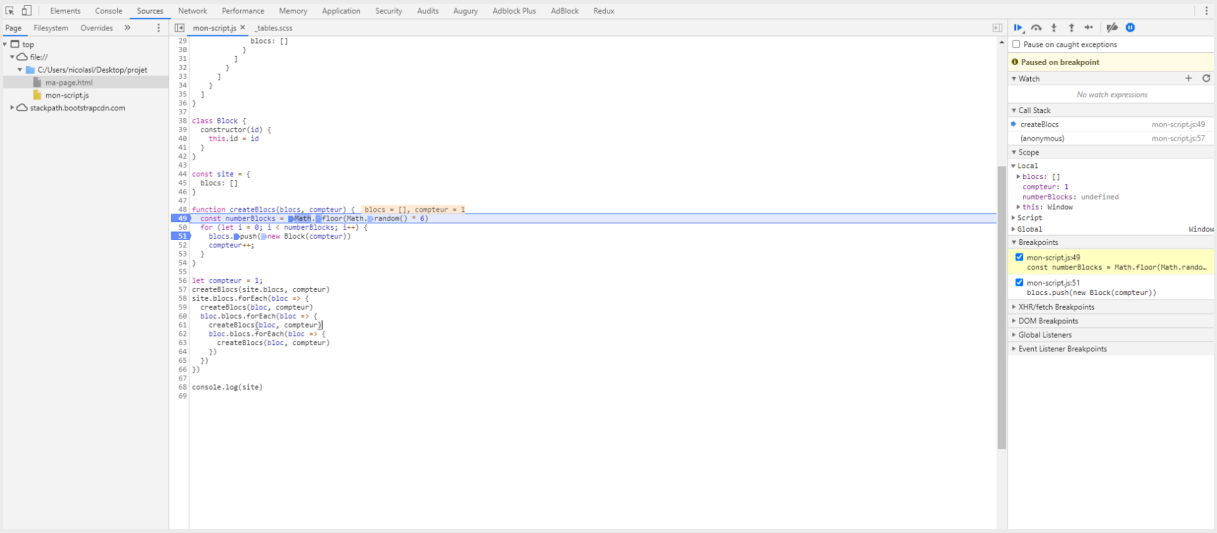
Exemple **Un point d'arrêt dans la console Google Chrome**

Pour créer un point d'arrêt, il faut commencer par ouvrir la console de votre navigateur. La section source est divisée en 3 parties.

À gauche se situent les fichiers du site (HTML, JavaScript, etc..). Cliquons sur le fichier JavaScript que l'on souhaite tester.

Au centre se trouve le code du fichier sélectionné. Les lignes du code sont numérotées. Pour placer le point d'arrêt, cliquons sur le numéro de la ligne à tester.

À droite se trouve le résumé du débug (nombre de points d'arrêt, scope, etc.). Le menu `scope` permet de voir les valeurs des variables au moment où le point d'arrêt s'exécute.



Conseil

Bon nombre de débogueurs permettent l'affichage de la pile des appels (*callstack*) : il peut être utile de l'utiliser pour remonter la source de l'appel à l'instruction.

Remarque

Si le bug a été introduit par une nouvelle version du code, il peut être intéressant de jouer le cas de test sur une version précédente du code pour vérifier si le bug n'était pas déjà présent. S'il n'est pas levé, essayez avec le commit suivant. Et ainsi de suite, jusqu'à identifier la modification responsable de l'erreur.

Si l'on n'a pas la possibilité de placer des points d'arrêt (par exemple, en environnement d'intégration) ou en cas de bug impossible à reproduire, les logs seront nos meilleurs alliés.

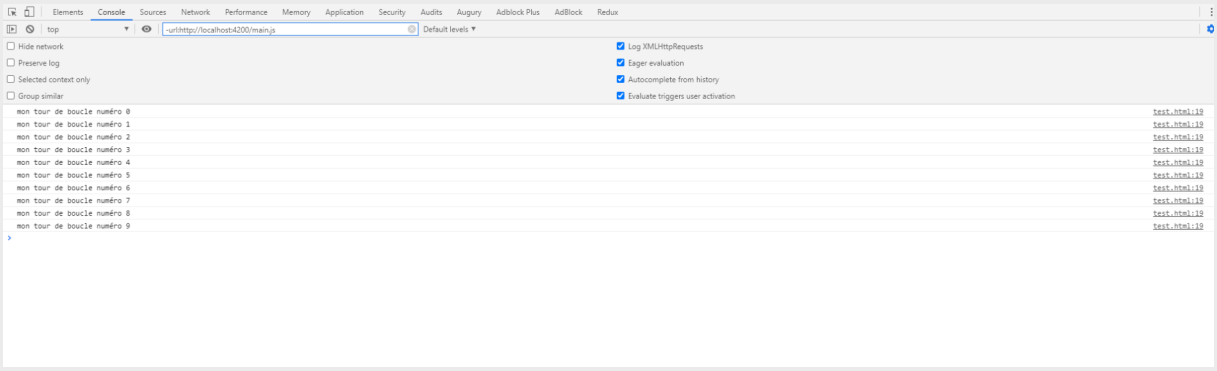
Attention cependant, l'ajout de logs peut modifier le comportement du programme et donc influer sur le bug !

Complément **console.log**

Une autre stratégie de débogage utilisée est l'utilisation de la méthode `console.log()`. Cette méthode retourne en console le résultat de la variable passée en paramètre. C'est une méthode très pratique dans le cas d'un débogage au sein d'une boucle, par exemple. En effet, elle permet d'avoir un résultat visuel de la liste des valeurs retournées ou affectées à chaque tour de boucle.

Exemple **Une boucle simple**

```
for (let counter = 0; counter < 10; counter++) {
  const text = 'mon tour de boucle numéro ' + counter;
  console.log(text)
}
```



Attention **Toujours supprimer les console.log()**

Il ne faut surtout pas oublier de supprimer les `console.log()` du code. Lorsqu'il ira en production et sera accessible par des utilisateurs, ceux-ci pourront voir les données retournées par les `console.log()` du code. Si des codes sensibles ont été testés, comme un mot de passe administrateur par exemple, il sera affiché en clair à tous les utilisateurs. Cela peut engendrer des failles de sécurité.

Corriger un bug

Une fois le bug identifié, reproductible et localisé, il faut le corriger en respectant le niveau de qualité du code attendu sur le projet.

Parfois, par manque de temps, le développeur est obligé de faire une correction rapide qui ne respecte pas le niveau de qualité, ou qui laisse sciemment des pans non fonctionnels. La décision ne doit pas être prise uniquement par le développeur, mais il devra être capable d'argumenter et de donner les limites du correctif. Ce *quickfix* doit être **temporaire**, **facilement identifiable** dans le code, **tracé** dans un gestionnaire de tâches, et surtout cela doit rester **extrêmement rare** !

Dans tous les cas, une fois le bug corrigé, n'hésitez pas à partager votre expérience avec l'équipe projet. Cela contribuera à l'amélioration des processus et de la qualité du code en général.

Complément

Le débogage est un des aspects du métier du développeur les plus exigeants. Il y a une attitude à tenir pour favoriser le travail :

- Toutes les parties du code sont potentiellement responsables de l'erreur,
- Il ne faut pas négliger les plus petites erreurs, une faute de frappe peut entraîner d'énormes problèmes,
- Avant de démarrer le *debug*, il faut s'assurer de ne pas avoir de modifications du code en cours,
- Il faut aménager suffisamment de temps pour pouvoir se concentrer sans être dérangé,
- **Le partage d'expérience est fondamental.**

Syntaxe **À retenir**

- La présence de bugs dans un code est inévitable. De plus, le débogage est un des aspects les plus exigeants du métier de développeur. Il demande beaucoup de rigueur et de méthode.
- Mais, si ces étapes sont respectées, il sera toujours possible de trouver la source du problème.

VII. Exercice : Appliquez la notion

Question

À partir d'une liste de températures prises chaque jour durant 1 mois, nous voulons répertorier les jours où la température a dépassé 20 °C. Pour cela, nous souhaitons stocker ces températures dans un tableau.

Le code suivant retourne une erreur en console. À partir du cours précédent, analysez le code et trouvez l'erreur.

```
const listTemperature = {
  month: 'January',
  tempPerDay : [
    {day: 1, temp: 15},
    {day: 2, temp: 10},
    {day: 3, temp: 14},
    {day: 4, temp: 20},
    {day: 5, temp: 18},
    {day: 6, temp: 17},
    {day: 7, temp: 15},
    {day: 8, temp: 16},
    {day: 9, temp: 15},
    {day: 10, temp: 9},
    {day: 11, temp: 10},
    {day: 12, temp: 13},
    {day: 13, temp: 14},
    {day: 14, temp: 11},
    {day: 15, temp: 12},
    {day: 16, temp: 16},
    {day: 17, temp: 17},
    {day: 18, temp: 14},
    {day: 19, temp: 15},
    {day: 20, temp: 11},
    {day: 21, temp: 11},
    {day: 22, temp: 10},
    {day: 23, temp: 19},
    {day: 24, temp: 22},
    {day: 25, temp: 18},
    {day: 26, temp: 17},
    {day: 27, temp: 16},
    {day: 28, temp: 13},
    {day: 29, temp: 15},
    {day: 30, temp: 20},
    {day: 31, temp: 22}
  ]
}
```

```
let tableTemp;

for (let counter = 0; counter < listTemperature.tempPerDay.length; counter++) {
  if (listTemperature.tempPerDay[counter].temp > 20) {
    tableTemp.push(listTemperature.tempPerDay[counter].temp);
  }
```

```
}

console.log(tableTemp)
```

Indice :

Lorsque la console retourne une erreur, il est fréquent qu'elle indique la ligne où s'est produit l'erreur. En cliquant dessus, vous atteindrez la ligne incriminée dans l'onglet Source.

VIII. Auto-évaluation

A. Exercice final

Exercice

Exercice

Comment s'appelle la stratégie la plus employée pour établir un diagnostic ?

- Le Front-tracking
- Le Back-tracking
- Le Deep-tracking
- Le Script-tracking

Exercice

En quoi consiste-t-elle?

- Ajouter un ou plusieurs points d'arrêt et remonter jusqu'à l'erreur
- Faire lire son code par un collaborateur
- Tout effacer et tout recommencer
- Ajouter des `console.log()` à différents endroits du code

Exercice

Lorsque l'on fait une réparation rapide par manque de temps et qui ne correspond pas aux standards de code de l'application, les bonnes pratiques sont de...

- [ ] Mettre un commentaire avec le préfixe "TODO"
- [ ] En parler avec l'équipe et le noter dans le ticket de la feature
- [ ] Tant que ça fonctionne, pas besoin de remonter la réparation
- [ ] Passer à autre chose et garder le refactoring de ce morceau de code dans un coin de sa tête

Exercice

Voila 1 heure que vous essayez de résoudre un bug. Vous n'y arrivez pas. Que faire ?

- [ ] Abandonner, c'est trop difficile
- [ ] Demander de l'aide à un collaborateur
- [ ] Prendre une pause ou passer à autre chose et y revenir plus tard
- [ ] Faire des tests empiriques en espérant trouver par miracle

Exercice

Dans la console du navigateur, quel est l'onglet qui permet d'afficher le code HTML et CSS ?

Exercice

Qu'est-ce qu'un point d'arrêt ?

- Un opérateur JavaScript
- Un point qui permet de mettre le code en pause
- Un point qui permet de voir les requêtes HTTP

Exercice

Quel mot-clé est utilisé en JavaScript pour mettre le code en pause ?

Exercice

En utilisant le mot-clé `debugger` dans votre code, il faut que la console navigateur soit ouverte pour que le code se mette en pause.

- Vrai
- Faux

Exercice

Quelle est l'extension console de debugger utilisé pour React ?

Exercice

Quelle est l'extension console de debugger utilisée pour Angular ?

B. Exercice : Défi

Dans le cadre d'un développement, un bug est survenu dans un algorithme. On vous demande de réparer ce bug.

Question 1

À partir de ce code, vous devrez retourner une liste d'objets au format **`{letter : 'A', names : ['Amandine', 'Armand']}`**.

Ce code trie chaque nom du tableau `data` par ordre alphabétique, puis retourne un tableau (`dataSorted`) d'objets ayant comme propriétés `letter` et `names`.

Copiez ce code dans un fichier HTML et exécutez-le dans votre navigateur.

```
<!doctype html>
<html lang="fr">
  <head>
    <meta charset="utf-8">
    <title>Tri par ordre alphabétique</title>
  </head>
  <body></body>
  <script>
  // tableau de noms
  const data = ['Paul', 'Jean', 'Marie', 'Nicolas', 'Julie', 'Lola', 'Martin', 'Armand',
'Amandine'];

```

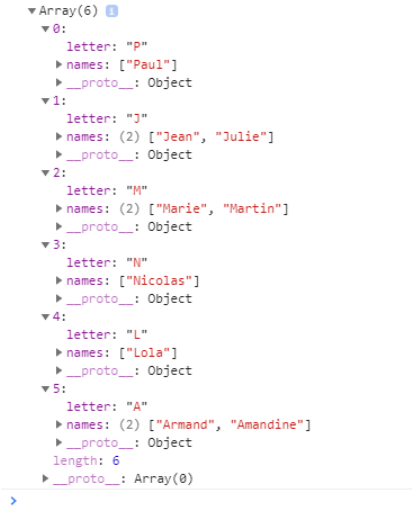
```
  // tableau de stockage de l'objet {letter: '', names: ['']}
  const dataSorted = [];

  // tri du tableau data par ordre alphabétique
  data.sort(function(a, b) {
    return a - b;
  });

  // POUR CHAQUE nom du tableau data
  data.forEach(function(name) {
    // On recherche s'il existe une valeur de la propriété letter égale à la premiere lettre du nom
    // Si une des valeur de la propriété letter est égale à la premiere lettre du nom on retourne l'index sinon on retourne -1
    const indexLetter = dataSorted.findIndex(function(objet) {
      return name.substring(0, 1).toLocaleLowerCase() === objet.letter.toLocaleLowerCase();
    });
    // Si l'index est différent de -1
    if (indexLetter !== -1) {
      // On ajoute le nom au tableau de la propriété names correspondant à l'index de dataSorted
      dataSorted[indexLetter].names.push(name);
    // Sinon
    } else {
      // On ajoute l'objet {letter: name.substring(0, 1), names: [name]} à dataSorted
      dataSorted.push({letter: name.substring(0, 1), names: [name]});
    }

  })
  console.log(dataSorted)
  </script>
</html>
```

Le résultat final devra être :

```
/*
[
  {
    letter: 'A',
    names: ['Amandine', 'Armand']
  },
  {
    letter: 'J',
    names: ['Jean', 'Julie']
  },
  {
    letter: 'L',
    names: ['Lola']
  },
  {
    letter: 'M',
    names: ['Marie', 'Martin']
  },
  {
    letter: 'N',
    names: ['Nicolas']
  },
  {
    letter: 'P',
    names: ['Paul']
```

```
  },
]
*/
```

Lorsqu'on exécute le code pour la première fois, la console nous retourne cet objet. On remarque que les lettres ne sont pas triées par ordre alphabétique.



Dans un premier temps, placez le point d'arrêt sur la ligne 27 de votre console, exécutez le code et affichez les informations sur l'objet `data`. Les items de l'objet devraient être positionnés dans l'ordre alphabétique. Est-ce la cas ?

Indice :

Chaque ligne de code est commentée pour expliquer ce que fait la fonction.

Ce code comporte une seule erreur, que vous pourrez trouver en lisant la documentation et en utilisant les points d'arrêt.

Question 2

Les commentaires dans le code nous indiquent que la méthode qui trie par ordre alphabétique est `sort()`. On peut donc en déduire qu'il y a un problème avec l'implémentation de la méthode. Recherchez dans la documentation MDN la méthode `sort()`. À partir des informations fournies par celle-ci, trouvez :

- Pourquoi la méthode `sort()` ne fonctionne pas dans notre cas ?
- Que faut-il faire pour que la méthode fonctionne ?

Question 3

Maintenant que nous avons compris l'erreur, nous pouvons remplacer le code erroné. Modifiez la méthode `sort()` pour qu'elle trie par ordre alphabétique, puis exécutez le code.

Solutions des exercices

Exercice p. Solution n°1

Il peut y avoir plusieurs fois le même numéro dans le tableau des numéros gagnants et le tableau des nombres complémentaires, alors qu'il ne doit pas pouvoir y avoir de doublons.

Exercice p. Solution n°2

```
// retourne un nombre aléatoire entre un minimum et un maximum
function randomInInterval(min, max){
  return Math.floor(Math.random() * (max - min + 1)) + min;
}
```

`Math.floor()`[1]

`Math.random()`[2]

```
// ajoute un nombre dans une table et supprime ce nombre dans le tableau de reférence
function addToTable(table, tableRef) {
  const index = randomInInterval(0, tableRef.length - 1)
  table.push(tableRef[index]);
  tableRef.splice(1, index);
}
```

`Array.push()`[3]

`Array.splice()`[4]

```
// boucle sur le tableau de résultat tant que la taille du tableau est inférieure à size
function tirage(size, table, tableRef) {
  while (table.length < size) {
    addToTable(table, tableRef)
  }
}
```

Exercice p. Solution n°3

```
const testRef = [1, 2, 3];
const testToGet = [];

const index = randomInInterval(0, testRef.length - 1);
testToGet.push(testRef[index]);
testRef.splice(1, index);

console.log(testRef);
console.log(testToGet);
```

On remarque que la taille du tableau `testRef` n'est jamais la même. Or, précédemment, on a vu que la méthode `splice()` supprimait un élément du tableau.

Il y a donc un problème avec la méthode `splice()`.

Exercice p. Solution n°4

[1] https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Math/floor
[2] https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Math/random
[3] https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Array/push
[4] https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Array/splice

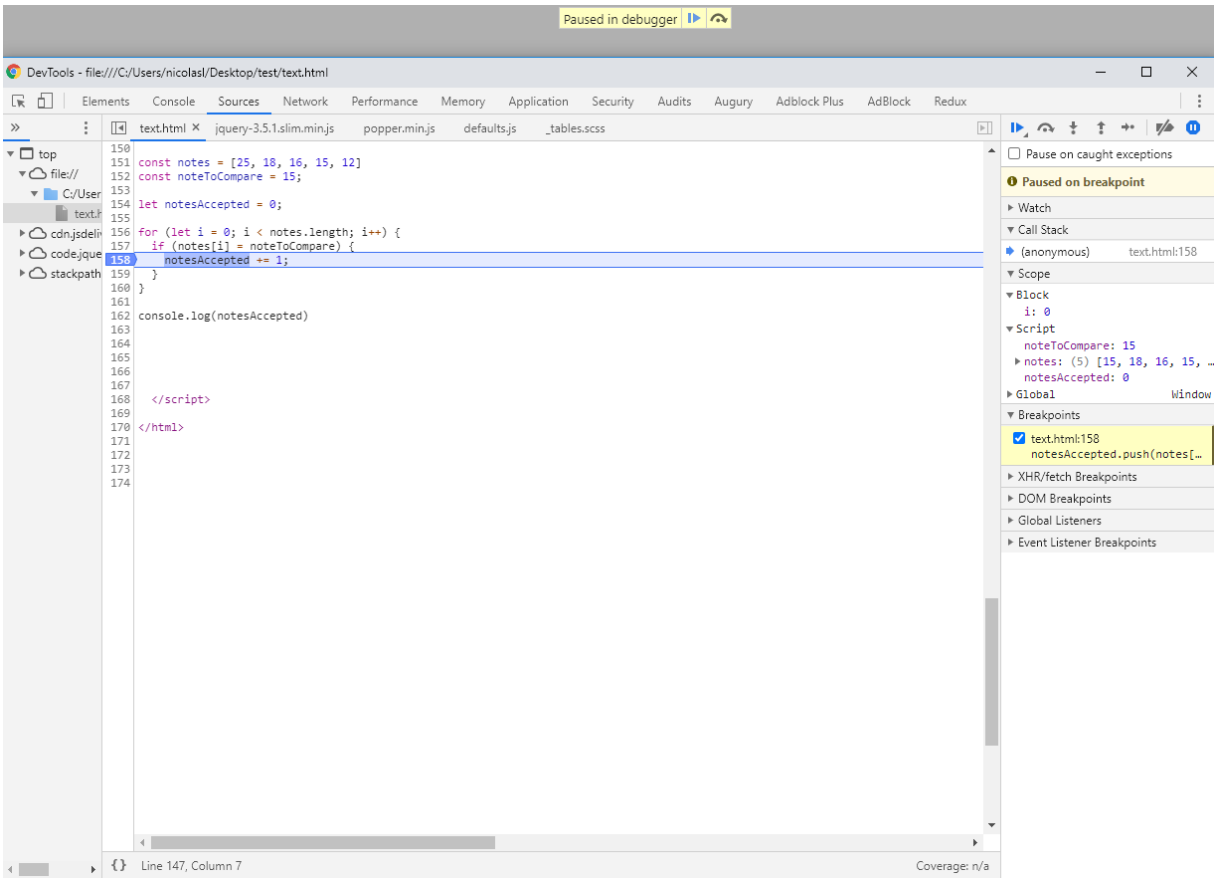
Dans la fonction, le rôle de la méthode `splice()` est de supprimer un élément à l'index `x` . Si l'on regarde la documentation, on peut lire que, pour supprimer un élément à l'index 5, la méthode s'écrit ainsi : `array.splice(5, 1)`. Notre code fait l'inverse : `array.splice(1, 5)`.

L'erreur vient donc d'une faute d'inattention, on a inversé les deux paramètres de la méthodes `splice()`.

```
const winNumbers = [];
const complementaryNumbers = []

function randomInInterval(min, max){
  return Math.floor(Math.random() * (max - min + 1)) + min;
}

function addToTable(table, tableRef) {
  const index = randomInInterval(0, tableRef.length - 1)
  table.push(tableRef[index]);
  tableRef.splice(index, 1);
}

function tirage(size, table, tableRef) {
  while (table.length < size) {
    addToTable(table, tableRef)
  }
}

tirage(5, winNumbers, listeNumber);
tirage(2, complementaryNumbers, listeNumber);

console.log(winNumbers, complementaryNumbers);
```

Exercice p. Solution n°5

Plaçons un point d'arrêt dans le navigateur et avançons pas-à-pas. On se rend compte que l'on atteint 5 fois le point d'arrêt, ce qui signifie que la condition est vérifiée 5 fois.



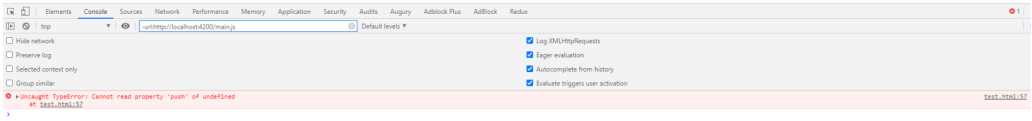
Le problème vient donc très certainement de la condition. Lorsqu'on regarde celle-ci, on se rend compte qu'une erreur d'inattention a été commise. L'opérateur = est un opérateur d'égalité, alors qu'il faudrait un opérateur de comparaison ===.

```
const notes = [25, 18, 16, 15, 12]
const noteToCompare = 15;

let notesAccepted = 0;

for (let i = 0; i < notes.length; i++) {
  if (notes[i] === noteToCompare) {
    notesAccepted += 1;
  }
}

console.log(notesAccepted) // 1
```

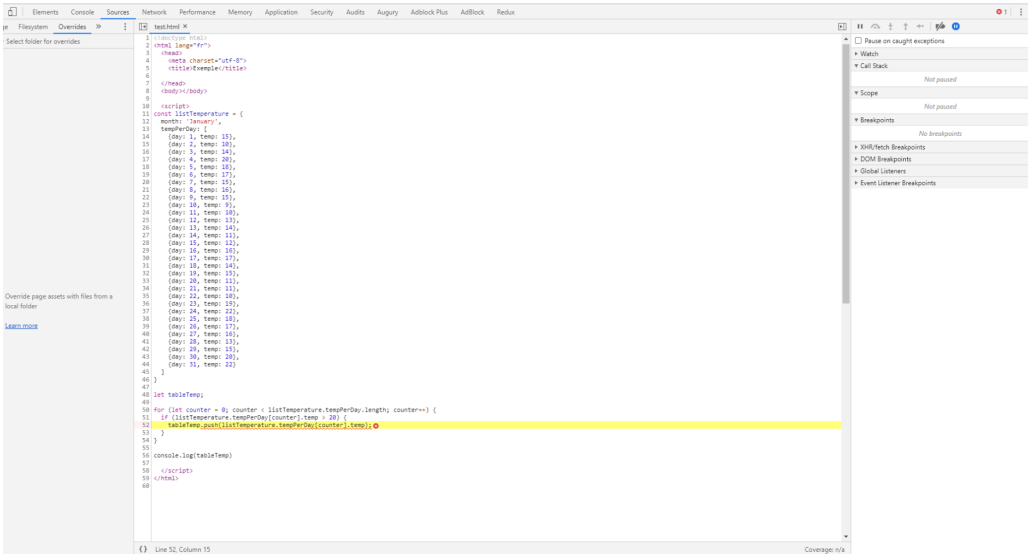
Exercice p. Solution n°6

Lorsque vous exécutez le code dans votre navigateur, ouvrez la console. Celle-ci retourne une erreur : `test.html:52 Uncaught TypeError: Cannot read property 'push' of undefined`.

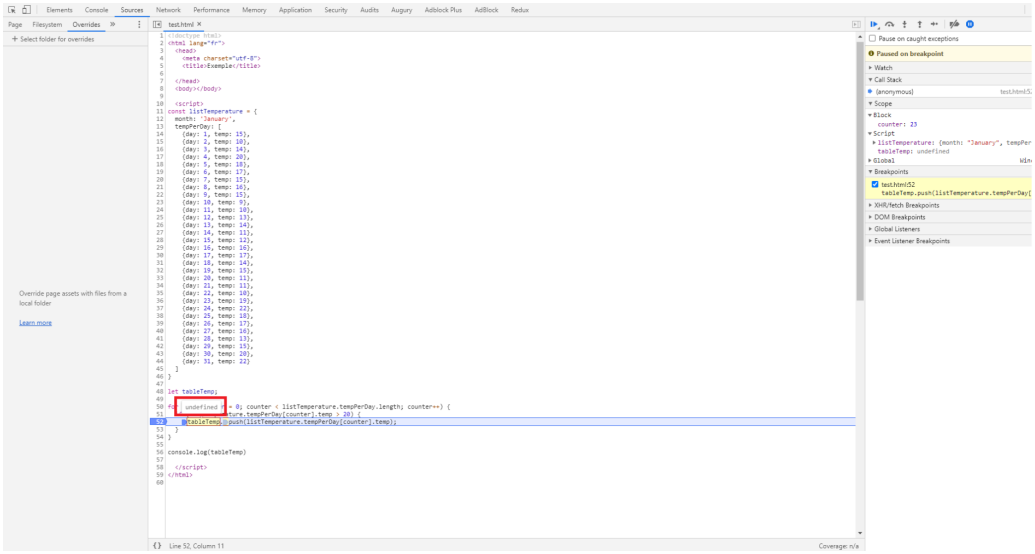
C'est un premier indice qui nous explique que l'on ne peut pas ajouter un élément dans un tableau qui est `undefined`.



À droite de l'erreur, on peut voir la ligne qui pose problème. En cliquant dessus, nous sommes redirigés vers la ligne en question dans l'onglet source.



À présent, plaçons un point d'arrêt pour y voir plus clair. Lorsque nous passons le curseur de la souris au-dessus de la variable `tableTemp`, une fenêtre nous indique que la variable est `undefined`.



Cela signifie que JavaScript ne reconnaît pas `tableTemp` comme un tableau et que cette variable n'est pas initialisée. Il faut donc initialiser la variable `tableTemp` pour résoudre le bug.

```
const tableTemp = []; //initialisation de tableTemp

for (let counter = 0; counter < listTemperature.tempPerDay.length; counter++) {
  if (listTemperature.tempPerDay[counter].temp > 20) {
    tableTemp.push(listTemperature.tempPerDay[counter].temp);
  }
}

console.log(tableTemp)
```

Exercice p. 16 Solution n°7

Exercice

Comment s'appelle la stratégie la plus employée pour établir un diagnostic ?

- ○ Le Front-tracking
- ◉ Le Back-tracking
- ○ Le Deep-tracking
- ○ Le Script-tracking

Exercice

En quoi consiste-t-elle?

- ◉ Ajouter un ou plusieurs points d'arrêt et remonter jusqu'à l'erreur
- ○ Faire lire son code par un collaborateur
- ○ Tout effacer et tout recommencer
- ○ Ajouter des `console.log()` à différents endroits du code

Exercice

Lorsque l'on fait une réparation rapide par manque de temps et qui ne correspond pas aux standards de code de l'application, les bonnes pratiques sont de...

- ☑ Mettre un commentaire avec le préfixe "TODO"
- ☑ En parler avec l'équipe et le noter dans le ticket de la feature
- ☐ Tant que ça fonctionne, pas besoin de remonter la réparation
- ☐ Passer à autre chose et garder le refactoring de ce morceau de code dans un coin de sa tête

Exercice

Voila 1 heure que vous essayez de résoudre un bug. Vous n'y arrivez pas. Que faire ?

- ☐ Abandonner, c'est trop difficile
- ☑ Demander de l'aide à un collaborateur
- ☑ Prendre une pause ou passer à autre chose et y revenir plus tard
- ☐ Faire des tests empiriques en espérant trouver par miracle

Exercice

Dans la console du navigateur, quel est l'onglet qui permet d'afficher le code HTML et CSS ?

Elements
Exercice

Qu'est-ce qu'un point d'arrêt ?

- ○ Un opérateur JavaScript
- ◉ Un point qui permet de mettre le code en pause
- ○ Un point qui permet de voir les requêtes HTTP

Exercice

Quel mot-clé est utilisé en JavaScript pour mettre le code en pause ?

debugger
Exercice

En utilisant le mot-clé `debugger` dans votre code, il faut que la console navigateur soit ouverte pour que le code se mette en pause.

- ◉ Vrai
- ○ Faux

Exercice

Quelle est l'extension console de debugger utilisé pour React ?

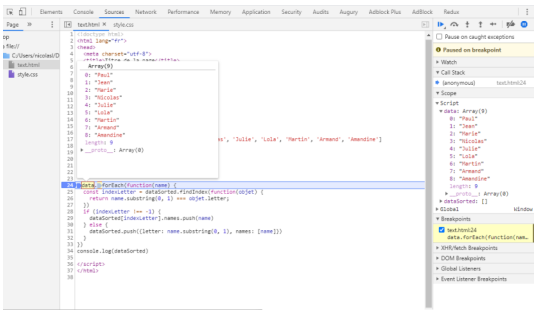
React developer tools
Exercice

Quelle est l'extension console de debugger utilisée pour Angular ?

Augury

Exercice p. Solution n°8

En plaçant le curseur sur `data`, nous obtenons la valeur de la constante. Celle-ci est aussi visible dans la partie `Scope` à la droite de votre console. Cette valeur nous confirme que la fonction ne marche pas. Les noms ne sont pas triés par ordre alphabétique. Il faut donc regarder la documentation.



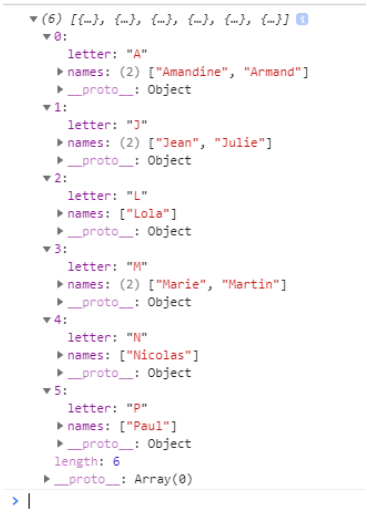
Exercice p. Solution n°9

À première vue, le code devrait marcher. Or, si l'on parcourt la documentation, on se rend compte que la méthode `sort()` trie des nombres et non des lettres. Pour résoudre ce problème, la documentation nous fournit une solution.

Exercice p. Solution n°10

Remplaçons donc l'intérieur de la fonction `sort()` par : `return a.localeCompare(b);`.

La console nous retourne bien le tableau trié par ordre alphabétique. Nous avons résolu le bug.



Remarque

Ce code est écrit en ES5 afin que vous puissiez vous repérer plus facilement dans la documentation. Toutefois, vous rencontrerez souvent la syntaxe ES6. Voila ce code écrit en ES6 :

```
const data = ['Paul', 'Jean', 'Marie', 'Nicolas', 'Julie', 'Lola', 'Martin', 'Armand', 'Amandine']
const dataSorted = [];
data.sort((a, b) => a.localeCompare(b));
data.forEach((name) => {
  const indexLetter = dataSorted.findIndex((objet) => name.substring(0, 1).toLocaleLowerCase() === objet.letter.toLocaleLowerCase());
  indexLetter !== -1 ? dataSorted[indexLetter].names.push(name) : dataSorted.push({letter: name.substring(0, 1), names: [name]});
})
```

Il est plus concis, mais moins facilement déboguable. Il faudra que vous appreniez les différentes syntaxes JavaScript (ES5 et ES6).