

Cours R2.03 - Débogage

R2.03 – Débogage

Étapes du développement :

Écriture du code

Compilation : mon programme est-il exempt d'erreurs de syntaxe ?

Exécution, tests fonctionnels : mon programme fait-il bien ce que je veux qu'il fasse ?

Si ce n'est pas le cas, il faut trouver les erreurs de conception en relisant le code

Deux possibilités peuvent aider à « comprendre ce qui ne va pas »

R2.03 - Débogage

Solution 1 - Ajout d'affichages dans la console :

```
if (element < tableau[tableau.length]) {  
    console.info('je rentre dans le if');  
  
    while (i < tableau.length && !trouve) {  
        console.error(i);  
        if (tableau[i] === element) {  
            trouve = true;  
        }  
        i++;  
    }  
}
```

Méthodes de console : log, info, debug, warn, error :
dans un navigateur web, la couleur change selon la
méthode (error, warn, info)

R2.03 - Débogage

Inconvénients :

- pas toujours très lisible s'il y a beaucoup de `console.log`, notamment dans des boucles
- on constate ce qui est affiché, mais ça n'aide pas forcément à comprendre pourquoi on obtient ces messages
- peu utilisable lorsque le code à déboguer est exécuté sur un dispositif mobile (la console est noyée par des messages système)
- il faut faire le ménage, bien penser à enlever tous les messages avant de mettre son code en production

R2.03 - Débogage

Solution 2 - Exécution « pas à pas »

Plutôt que de laisser le programme s'exécuter librement, on peut ajouter des

points d'arrêts (breakpoints)

puis, une fois que le programme a atteint la ligne de code correspondante, on peut ensuite exécuter le code ligne par ligne

R2.03 - Débogage

Exemple d'utilisation

Écrire un programme qui cherche un élément dans une table triée et indique si on l'a trouvé → **main.ts**

Ici on cherche 64, qui n'est pas trouvé...

```
const element = 64;
const tableau = [10, 64, 85, 128];
let trouve = false;
if (element < tableau[tableau.length]) {
  let i = 0;
  while (i < tableau.length && !trouve) {
    if (tableau[i] === element) {
      trouve = true;
    }
    i++;
  }
}
if (trouve) {
  console.log('trouvé !');
}
```

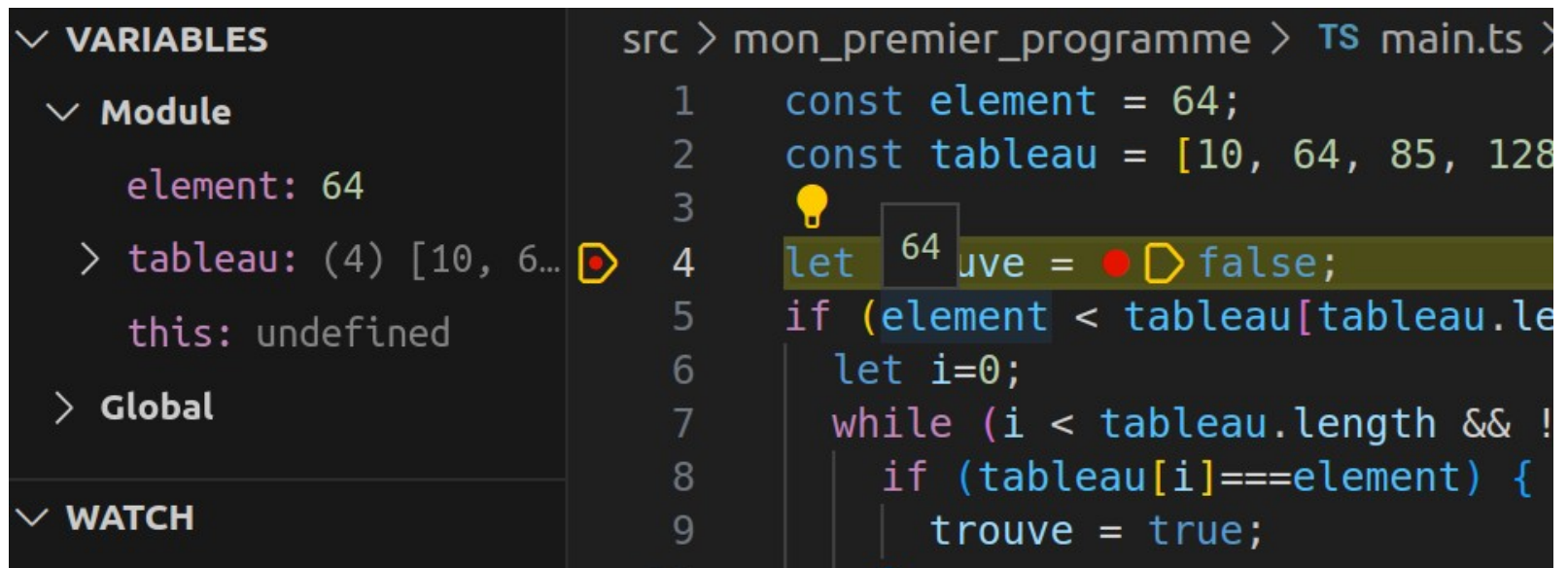
R2.03 - Débogage

Exemple d'utilisation

Exécution pas à pas : placement d'un breakpoint puis :

F5 et choisir « Deno » dans « Select debugger »

Constat : le programme ne rentre pas dans le `if`



The screenshot shows the Deno debugger interface. On the left, the 'VARIABLES' panel is expanded, showing the 'Module' scope with 'element: 64' and 'tableau: (4) [10, 64, 85, 128]'. The 'WATCH' panel is also visible. On the right, the code editor shows the file 'src > mon_premier_programme > TS main.ts'. The code is as follows:

```
1  const element = 64;
2  const tableau = [10, 64, 85, 128];
3
4  let trouve = false;
5  if (element < tableau[tableau.length - 1]) {
6    let i=0;
7    while (i < tableau.length && !trouve) {
8      if (tableau[i]===element) {
9        trouve = true;
```

A breakpoint is set on line 4, and the value of 'trouve' is shown as 'false' in the variable watch. A yellow lightbulb icon is visible next to line 3.

R2.03 - Débogage

Exemple d'utilisation

- Placement d'un point d'arrêt sur la ligne à problème

clic gauche

ou

clic droit, puis choix du type de point

```
1  const element = 64;
2  const tableau = [10, 64, 85, 128];
3  let trouve = false;
4  if (element < tableau[tableau.length]) {
5      // ...
6      while (i < tableau.length && !trouve) {
7          if (tableau[i] === element) {
8              trouve = true;
9          }
10     }
```

Ajouter un point d'arrêt
Ajouter un point d'arrêt conditionnel...
Ajouter un point de journalisation...

```
4  if (element < tableau[tableau.length]) {
5      let i = 0;
6      while (i < tableau.length && !trouve) {
7          if (tableau[i] === element) {
8              trouve = true;
9          }
10     }
```


R2.03 - Débogage

Exemple d'utilisation

- F5 pour exécuter jusqu'au 1^{er} point d'arrêt → stop sur la ligne

```
src > mon_premier_programme > TS main.ts > ...  
1  const element = 64;  
2  const tableau = [10, 64, 85, 128];  
3  let trouve = false;  
4  if (element < tableau[tableau.length]) {  
5      let i = 0;  
6      while (i < tableau.length 64 !trouve) {  
7          if (tableau[i] === element) {  
8              trouve = true;  
9          }  
10         i++;  
11     }  
12 }
```

VARIABLES

- Local
 - element: 64
 - exports: {}
 - module: Module {id: '.', path: '/ho...}
 - require: f require(path) {\n r...
 - tableau: (4) [10, 64, 85, 128]
 - trouve: false
- ESPION

R2.03 - Débogage

Exemple d'utilisation

- Partie gauche : le contenu des variables
- Partie « espion » pour évaluer des expressions

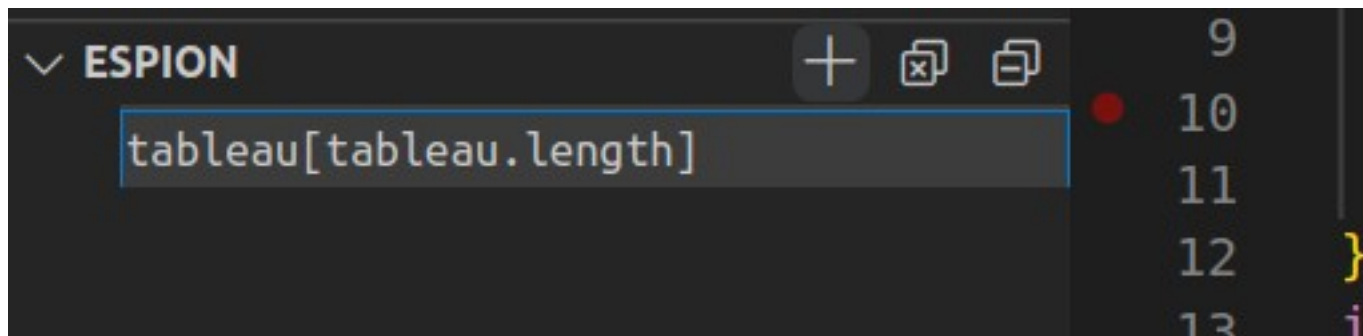
The screenshot shows the VS Code interface with a TypeScript file named `main.ts` open. The left sidebar displays the **VARIABLES** pane, which is divided into **Local** and **ESPION** sections. The **Local** section shows the current state of variables: `element` is 64, `exports` is an empty object, `module` is a Module object, `require` is a function, `tableau` is an array `[10, 64, 85, 128]`, and `trouve` is `false`. The **ESPION** section is currently empty. The main editor shows the source code of `main.ts` with line numbers 1 through 10. The code defines a constant `element` with value 64, a constant array `tableau` with values `[10, 64, 85, 128]`, and a variable `trouve` set to `false`. It then enters an `if` block where the condition `element < tableau[tableau.length]` is being evaluated. The value `64` is highlighted in the code, corresponding to the value of `element` shown in the variables pane. The code continues with a `while` loop and an `if` statement inside it.

```
src > mon_premier_programme > TS main.ts > ...  
1  const element = 64;  
2  const tableau = [10, 64, 85, 128];  
3  let trouve = false;  
4  if (element < tableau[tableau.length]) {  
5      let i = 0;  
6      while (i < tableau.length 64 !trouve) {  
7          if (tableau[i] === element) {  
8              trouve = true;  
9          }  
10         i++;  
11     }  
12 }
```

R2.03 - Débogage

Exemple d'utilisation

- Pourquoi on ne rentre pas dans le `if` ?
 - `element` vaut 64, que vaut `tableau[tableau.length]` ?
- clic sur « + » à droite d'« `ESPION` », et ajout de l'expression



R2.03 - Débogage

Exemple d'utilisation

- Exécution avec F5, arrêt sur la ligne 4
- `tableau[tableau.length]` : undefined

R2.03 - Débogage

Exemple d'utilisation

- Correction de l'accès au dernier élément, exécution
→ résultat OK

```
4  if (element < tableau[tableau.length - 1]) {  
5  let i = 0;  
6  while (i < tableau.length && !trouve) {  
7  if (tableau[i] === element) {
```

R2.03 - Débogage

Commandes possibles

- F5 pour exécuter jusqu'au prochain point d'arrêt
- F10 pour exécuter ligne par ligne, en restant dans le sous-programme courant
- F11 pour exécuter ligne par ligne, en descendant dans les fonctions appelées
- Maj F11 pour sortir d'un sous-programme
- (on retrouve ces options dans le menu « Exécuter » et dans la barre de debug)



R2.03 - Débogage

Point d'arrêt conditionnel

- Utile notamment dans une boucle, pour voir ce qui se passe lorsqu'on attend un certain résultat et que ce n'est pas ce qui arrive

```
6   while (i < tableau.length && !trouve) {  
e 7   if (tableau[i] === element) {  
    Expression  v  i===1  
8   }  
    trouve = true;
```

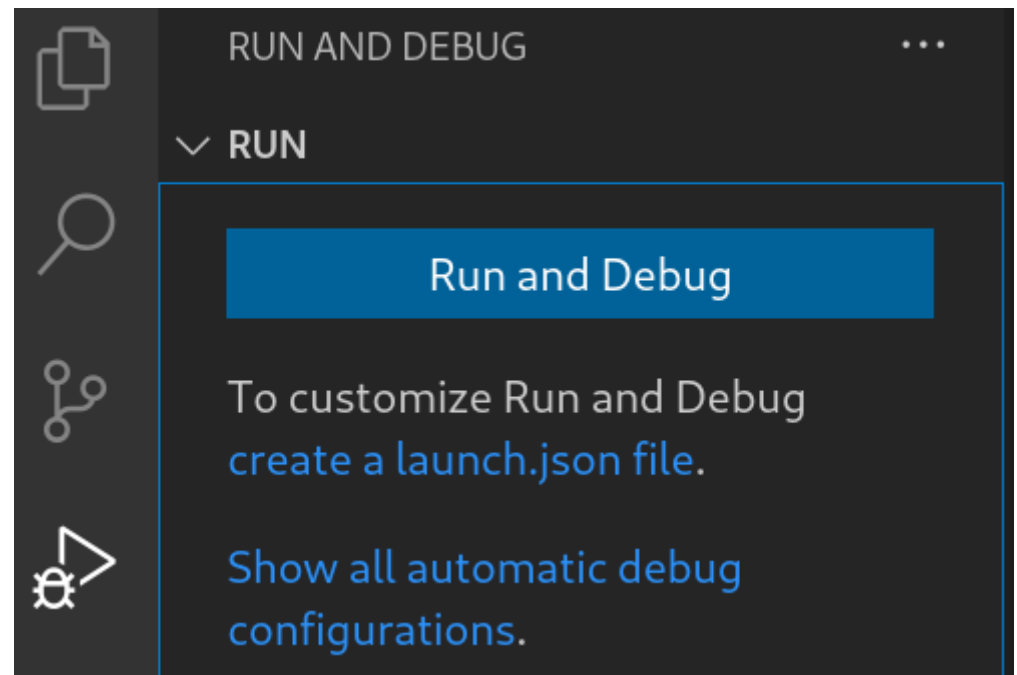
R2.03 - Débogage

Si votre programme fait une saisie au clavier

Par défaut, la saisie se fait dans la « debug console », ce qui n'est pas possible

Pour modifier cela, il faut créer un fichier de configuration.

- 1) Clic sur le bug à gauche, puis « create a launch.json file »
- 2) Sélectionner Deno



R2.03 - Débogage

3) Si nécessaire, modifier le nom du fichier à débogger (par défaut : **main.ts**)

4) **"console": "integratedTerminal"** pour que les saisies se fassent dans le terminal

```
"type": "node",
"program": "${workspaceFolder}/main.ts",
"cwd": "${workspaceFolder}",
"runtimeExecutable": "/home/laroche5/.der
"runtimeArgs": [
    "run",
    "--inspect-wait",
    "--allow-all"
],
"attachSimplePort": 9229,
"console": "integratedTerminal"
```