

R2.03 TD Debug

Ce TD donne lieu à un compte-rendu poussé sur GIT, dans le dépôt que vous avez utilisé pour le compte-rendu GIT. Vous répondrez à chaque question et indiquerez ce que vous avez appris à chaque exercice.

Vous pousserez également le code écrit à chaque exercice.

Faire des commits/push pendant la séance puis un tag en fin de séance de TD, et enfin un tag final pour le 30 mars dernier délai.

Exercice 1

Placer un point d'arrêt dans le programme suivant pour pouvoir suivre les valeurs successives de la variable `j`.

```
function deb1(): number {  
  let j = 200;  
  for (let i = 0; i < 10; i++) {  
    j -= 10;  
  }  
  return j;  
}
```

Exercice 2

Ajouter dans la fonction une déclaration de tableau `tab`. Dans la boucle, affecter `j` à `tab[i]`. Visualiser à l'aide de l'outil de debug le remplissage du tableau.

Exercice 3

En plaçant un ou plusieurs points d'arrêts, indiquer les valeurs successives des variables `x` et `y`.

```
function deb2(x: number, y: number): void {  
  const tmp = y;  
  y = x;  
  x = tmp;  
  console.log(x, y);  
}  
  
let x = 10;  
let y = 20;  
deb2(x, y);  
console.log(x, y);
```

En déduire un nom parlant pour la fonction `deb2`, et comprendre pourquoi cela ne fonctionne pas comme le programmeur aurait voulu.

Exercice 4

Faire de même pour les valeurs du tableau `xy`.
Comprendre pourquoi « ça marche ».

```
function deb3(xy: number[]): void {  
  let tmp= xy[0];  
  xy[0] = xy[1];  
  xy[1] = tmp;  
  console.log(xy);  
}  
let xy= [10, 20];  
deb3(xy);  
console.log(xy);
```

Question subsidiaire : avant la fin de fonction `deb3`, ajouter `xy=[100, 200]`. Qu'est-ce qui est affiché par le dernier `console.log` et pourquoi ?

Exercice 5

Exécuter le programme suivant en plaçant les points d'arrêt comme indiqué.
Indiquer les états successifs de la mémoire, en montrant les différentes variables appelées 'j'. En déduire la portée des variables.

```
1  let j = 10;  
2  deb4();  
3  console.log(j);  
4  
5  function deb4(): number {  
6    let j = 25;  
7    for (let i = 0; i < 10; i++) {  
8      let j = 2 * i;  
9      console.log(j);  
10   }  
11   return j;  
12 }  
13
```

Exercice 6

Exécuter pas à pas le programme suivant. Que constatez-vous ?

```
let tab1 = [1, 2, 3];  
let tab2 = tab1;  
  
tab1[1] = 5;  
|  
console.log(tab1, tab2);
```

Faire de même avec le code suivant.

```
let tab1 = [1, 2, 3];  
let tab2 = [...tab1];  
  
tab1[1] = 5;  
  
console.log(tab1, tab2);
```

En déduire ce qui s'est passé en ligne 2 et la différence avec la ligne 2 du programme précédent.

Exercice 7

Exécuter pas à pas le programme suivant. Comprendre ce qu'il fait ; corriger le nom de la fonction.

```
deb7(5, 3);

function deb7(x: number, y: number): number {
  let val = 0;

  for (let i = 0; i < y; i++) {
    val += x;
  }
  return val;
}
```

Exécuter pas à pas en modifiant la première ligne : `deb7(3, 5);` ;

En déduire une optimisation de la fonction.

Puis exécuter pas à pas avec `deb7(-5, 3)`. Constatez-vous un problème ?

Faire de même avec `deb7(5, -3)`. Même question, corriger le programme en conséquence.

Afin de valider votre programme, exécutez-le avec les valeurs qui suivent, en affichant le résultat et les valeurs de x, y utilisées pour la boucle ; votre programme doit donner la bonne solution et être optimisé.

`[(0, 0), (5, 3), (3, 5), (-5, -2), (-2, -5),
(-74, 2), (-1, 75), (10, -25), (10, -3)]`

Utilisez le mode pas à pas si vous avez du mal à comprendre pourquoi votre code ne fonctionne pas comme vous le voudriez.

Exercice 8

Exécuter le programme suivant pas à pas. Bien observer les différents `return`.

Indiquer les états successifs de la mémoire. Corriger le nom de la fonction. Cette fonction donne-t-elle toujours des résultats satisfaisants ? Si non, proposer une correction la plus optimisée possible.

```
let n=5;
console.log(deb8(n));
function deb8(n: number): number {
  if (n===0) {
    return 1;
  }
  let valeur = n * deb8(n-1);
  return valeur;
}
```