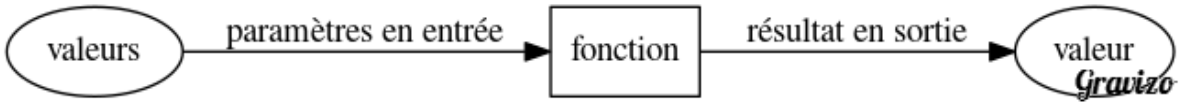
4: Fonctions

Introduction

Comme en mathématiques, une fonction transforme des paramètres (en "entrée") en une valeur de résultat (la "sortie"), lorsqu'elle est appelée. Avant de pouvoir l'appeler, on la définit par une suite d'instructions qui déterminera cette valeur de résultat, en fonction des paramètres qui lui seront passés.



Définir une fonction permet de regrouper des instructions JavaScript, afin qu'elles puissent être exécutées à différentes occasions, sans avoir à dupliquer le code correspondant.

Par exemple, sur le web, les fonctions JavaScript sont utilisées par le développeur pour définir le comportement que doit suivre le navigateur lorsque l'utilisateur effectue certaines actions (ex: saisie dans un champ, clic sur un bouton, soumission d'un formulaire).

Définition et appel de fonction

On définit une fonction de la manière suivante:

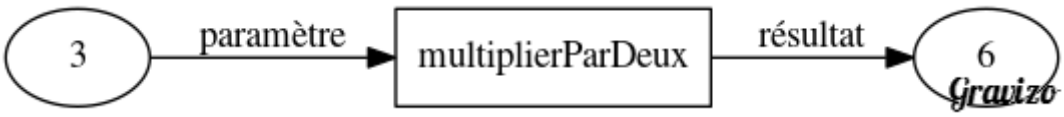
```
function nomDeLaFonction (parametre1, parametre2, parametre3 ...) {
  // instructions javascript
  // pouvant utiliser les paramètres parametre1, parametre2 et parametre3
  return resultat;
}
```

Par exemple:

```
function multiplierParDeux (nombre) {
  return nombre * 2;
}
```

Pour exécuter une fonction, il faut *l'appeler* en citant son nom, et en lui fournissant des valeurs pour chacun des paramètres entre parenthèses.

Par exemple:



```
var resultat = multiplierParDeux(3); // => le paramètre nombre vaut 3 => la
variable resultat vaudra 6
```

Comme pour une variable, l'appel à une fonction sera remplacé la valeur qu'elle renvoie, au moment de l'exécution du programme. Contrairement aux variables, cette valeur dépendra de la valeur des paramètres passés à la fonction.

Ainsi, il est possible de passer le résultat de l'appel d'une fonction en paramètre d'une fonction.

Exemple de substitution d'un appel de fonction par sa valeur de retour:


```
resultat = multiplierParDeux(multiplierParDeux(3)); // équivaut à:
resultat = multiplierParDeux(3 * 2); // qui équivaut à:
resultat = (3 * 2) * 2; // qui vaut finalement:
resultat = 12;
```

Et, avec une autre valeur passée en paramètre:

```
var resultat = multiplierParDeux(multiplierParDeux(4)); // équivaut à:
var resultat = multiplierParDeux(4 * 2); // qui équivaut à:
var resultat = (4 * 2) * 2; // qui vaut finalement:
var resultat = 16;
```

Importance de `return`

Quand on exécute une fonction depuis une console JavaScript, la valeur retournée par cette fonction est affichée dans la console. Il ne faut pas pour autant confondre le mot clé `return` et la fonction `console.log`.

En effet:

- `console.log` peut être appelée plusieurs fois depuis un même définition de fonction, mais chaque appel de fonction ne peut résulter qu'en une seule valeur de retour spécifiée par `return`,
- l'usage de `return` permet à l'appelant d'une fonction de disposer de la valeur résultante comme il le souhaite: il peut par exemple décider d'afficher cette valeur dans la console, ou dans un `alert`, ou même de la stocker dans une variable. Or, si la définition de cette fonction affiche la valeur résultante dans la console au lieu d'utiliser `return`, il sera impossible pour l'appelant de récupérer cette valeur résultante dans son programme.

Bugs et tests unitaires: comment tester une fonction

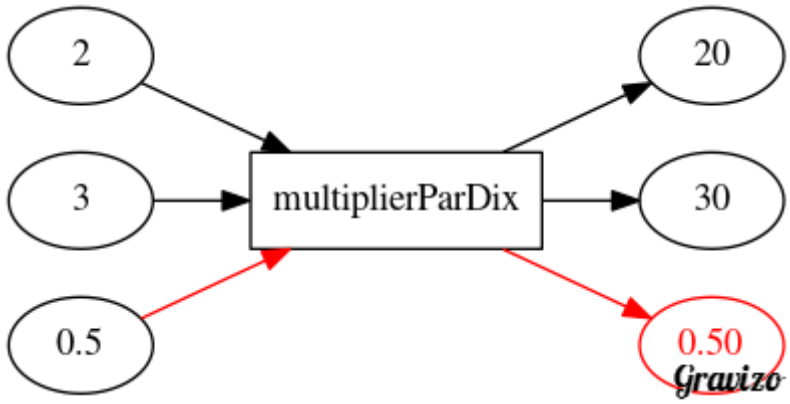
Appeler une fonction ajoute de l'incertitude et parfois de l'imprévisibilité au comportement du code, car cela revient à déléguer une fonctionnalité à une autre partie du code (la définition de la fonction appelée).

Afin de se rassurer sur le bon fonctionnement d'une fonction et éviter les *bugs*, il est important de tester les fonctions qu'on utilise.

Un *bug* est un comportement imprévu causant des anomalies et/ou l'interruption de l'exécution du programme. Il est généralement causé par une erreur d'implémentation ou une réalisation trop naïve (c.a.d. ne couvrant pas certains cas qui peuvent se produire).

Exemple d'implémentation naïve pouvant causer un bug:

```
function multiplierParDix (nombre) {
  return nombre + '0'; // on ajoute un zéro à la fin du nombre
}
multiplierParDix(2);   // => 20 => OK
multiplierParDix(3);   // => 30 => OK
multiplierParDix(0.5); // => 0.50 => BUG! on voulait obtenir 5 dans ce cas
```



Dans l'exemple ci-dessus, nous avons effectué trois tests d'appel de notre fonction `multiplierParDix`, et l'un d'eux nous a permis de détecter un bug dans notre fonction.

Afin de réduire le nombre de bugs potentiels d'une fonction, et donc de se rassurer sur son bon fonctionnement, il est important d'écrire et exécuter plusieurs tests unitaires, et penser intelligemment aux *cas limites*, les cas qui pourraient le plus probablement causer un bug.

Écrire un test unitaire pour une fonction consiste à:

- décrire un exemple d'usage de cette fonction, en précisant la valeur résultante attendue pour certaines valeurs de paramètres,
- implémenter l'appel de cette fonction, et comparer la valeur résultante à celle qui est attendue.

Lors de l'exécution du test unitaire, si la valeur de la comparaison détermine si la fonction fonctionne comme prévue sur l'exemple de ce test.

Par exemple, on pourrait définir les trois tests unitaires suivants pour valider notre fonction `multiplierParDix`:

```
multiplierParDix(2) === 20;  // => false (car '20' différent de 20)
multiplierParDix(3) === 30;  // => false (car '30' différent de 30)
multiplierParDix(0.5) === 5; // => false (car '0.50' différent de 0.5)
```

Avec la définition de la fonction `multiplierParDix` fournie plus haut, aucun de ces tests ne *passe*. C'est à dire que chaque test d'égalité sera `false`.

En revanche, les tests unitaires *passeront* avec la définition suivante de cette même fonction:

```
function multiplierParDix (nombre) {
  return nombre * 10;
}
multiplierParDix(2) === 20;  // => true => OK
multiplierParDix(3) === 30;  // => true => OK
multiplierParDix(0.5) === 5; // => true => OK
```

> À retenir: Un test unitaire est un exemple d'appel permettant de vérifier qu'une fonction se comporte comme prévu dans un cas donné, en comparant le résultat effectivement retourné au résultat qui devrait être retourné.

Valeur et affectation d'une fonction

Nous avons vu que l'appel d'une fonction consiste à mentionner son nom, suivi de paramètres exprimés entre parenthèses. Et que cet appel est remplacé par la valeur retournée par son exécution.

```
// définition de la fonction multiplierParDeux()
function multiplierParDeux (nombre) {
  return nombre * 2;
}

// appel de la fonction multiplierParDeux(), en passant le nombre 3 en paramètre
var resultat = multiplierParDeux(3);

// la valeur retournée par l'appel de la fonction (6) est affectée à résultat
```

Ainsi, `multiplierParDeux(3)` est remplacé par sa valeur de retour: `6`, après l'exécution de la fonction `multiplierParDeux` à laquelle on a passé la valeur littérale `3` comme valeur du paramètre appelé `nombre`.

Pour rappel, une variable Javascript est remplacée par la dernière valeur qui lui a été affectée. Ainsi, si la valeur `6` a été affectée à la variable `maVariable` à l'aide de l'instruction `maVariable = 6;`, les mentions suivantes de `maVariable` seront remplacée par sa valeur `6`.

En Javascript, une fonction est une valeur, au même titre qu'un nombre ou une chaîne de caractères. Elle peut donc aussi être attribuée à une variable.

Ainsi, il est possible d'affecter la fonction `multiplierParDeux` à la variable `maVariable`:

```
maVariable = multiplierParDeux;
```

...et de l'appeler de la manière suivante:

```
maVariable(3); // => retourne la valeur 6;
```

Il est donc aussi possible d'affecter une fonction anonyme à une variable:

```
var multiplierParTrois = function (nombre) {
  return nombre * 3;
};
```

... ce qui est équivalent à écrire:

```
function multiplierParTrois (nombre) {
  return nombre * 3;
}
```