> Cette page a été traduite à partir de l'anglais par la communauté. Vous pouvez également contribuer en rejoignant la communauté francophone sur MDN Web Docs.

Contrôle du flux d'instructions et gestion des erreurs

JavaScript supporte nativement un ensemble d'instructions. Ces instructions permettent de définir les logiques des algorithmes, le flux des informations, etc. Ce chapitre fournit un aperçu sur le fonctionnement de ces différentes instructions JavaScript.

Les Références de JavaScript contiennent plus de détails sur les différentes instructions décrites dans ce chapitre.

Toute expression est une instruction, voir la page Expressions et opérateurs pour plus d'informations sur les expressions. En JavaScript, le point-virgule (;) est utilisé afin de séparer des instructions dans le code.

Les blocs d'instruction

L'instruction la plus simple est l'instruction de bloc qui permet de regrouper des instructions. Un bloc est délimité par une paire d'accolades :

JS

```
{
   instruction_1;
   instruction_2;
   ⋮
   instruction_n;
}
```

Exemple

Les instructions de blocs sont souvent utilisées avec les instructions conditionnelles et itératives telles que `if`, `for`, `while`.

```
JS
while (x < 10) {
  x++;
}
```

Ici, `{ x++; }` représente le bloc.

> **Note :** En JavaScript, avant ECMAScript 2015 (aussi appelé ES6), les blocs **n'introduisaient pas de nouvelles portées**. Les variables introduites dans le bloc avec l'instruction `var` font partie de la portée de la fonction englobante ou du script. Les effets de leur définition persistent en dehors du bloc. Les blocs seuls utilisés avec `var` (et non `let`) pourront laisser penser que ce bloc se comportera comme en C ou en Java. Par exemple :
>
> ```
> JS
> var x = 1;
> {
>   var x = 2;
> }
> console.log(x); // affichera 2
> ```
>
> Cella affichera 2 car l'instruction `var x` contenue dans le bloc fait partie de la même portée que l'instruction `var x` écrite avant le bloc. En C ou en Java, le code équivalent à cet exemple aurait produit 1.
>
> Cela a évolué avec ECMAScript 2015 (ES6). Les instructions `let` et `const` permettent de déclarer des variables dont la portée est celle du bloc courant. Voir les pages des références `let` et `const`.

Les instructions conditionnelles

Une instruction conditionnelle est un ensemble de commandes qui s'exécutent si une condition donnée est vérifiée. JavaScript possède deux instructions conditionnelles :

`if...else` et `switch`.

Instruction `if...else`

On utilise l'instruction `if` lorsqu'on souhaite exécuter une instruction si une condition logique est vérifiée (vraie). La clause `else` est optionnelle et permet de préciser les instructions à exécuter si la condition logique n'est pas vérifiée (l'assertion est fausse). Voici un exemple qui illustre l'utilisation de l'instruction `if` :

JS

```
if (condition) {
  instruction_1;
} else {
  instruction_2;
}
```

`condition` peut correspondre à n'importe quelle expression qui est évaluée à `true` (vrai) ou `false` (faux). Voir la page sur les booléens pour plus d'informations sur les évaluations qui fournissent les valeurs `true` ou `false`. Si la `condition` vaut `true`, `instruction_1` est exécutée, sinon `instruction_2` sera exécutée. `instruction_1` et `instruction_2` peuvent correspondre à n'importe quelle instruction, y compris d'autres instructions `if`.

Si on doit tester différentes conditions les unes à la suite des autres, il est possible d'utiliser `else if` pour lier les différents tests. On l'utilise de la façon suivante :

JS

```
if (condition_1) {
  instruction_1;
} else if (condition_2) {
  instruction_2;
} else if (condition_n) {
  instruction_n;
} else {
  dernière_instruction;
}
```

Afin d'exécuter plusieurs instructions, on peut les regrouper grâce aux blocs (`{ ... }`) vus précédemment. C'est une bonne pratique que de les utiliser, surtout si on imbrique

plusieurs instructions `if` les unes dans les autres:

Meilleure pratique

En général, il est bon de toujours utiliser des instructions de type bloc —*surtout* lorsqu'on imbrique des instructions `if` :

JS

```
if (condition) {
  instruction_1_exécutée_si_condition_vraie;
  instruction_2_exécutée_si_condition_vraie;
} else {
  instruction_3_exécutée_si_condition_fausse;
  instruction_4_exécutée_si_condition_fausse;
}
```

Attention à ne pas utiliser des instructions d'affectation dans les expressions conditionnelles. On peut, en effet, très facilement confondre l'affectation et le test d'égalité en lisant le code.

Voici un exemple de ce qu'il ne faut **pas** faire :

JS

```
if (x = y) {
  /* exécuter des instructions */
}
```

Ici, on ne teste pas si x vaut y, on affecte la valeur de y à x ! Si vous devez à tout prix utiliser une affectation dans une expression conditionnelle, une bonne pratique sera d'ajouter des parenthèses en plus autour de l'affectation. Par exemple :

JS

```
if ((x = y)) {
  /* exécuter des instructions */
}
```

Valeurs équivalentes à false dans un contexte booléen (*falsy* values)

Les valeurs suivantes sont évaluées à `false` (également connues sous le nom de valeurs Falsy) :

- `false`
- `undefined`
- `null`
- `0`
- `NaN`
- la chaîne de caractères vide (`""`)

Les autres valeurs, y compris les objets, seront équivalents à `true` .

> **Attention :** Ne pas confondre les valeurs booléennes « primitives » `true` et `false` avec les valeurs créées grâce à un objet `Boolean` .
>
> Par exemple, on aura :
>
> JS
>
> ```
> var b = new Boolean(false);
>   if (b) // cette condition est bien vérifiée !
>   if (b === true) // cette condition n'est pas vérifiée !
> ```

Exemple

Dans l'exemple qui suit, la fonction `checkData` renvoie `true` si une chaîne de caractères mesure trois caractères. Sinon, elle affiche une alerte et renvoie `false` .

JS

```
function checkData(maChaîne) {
  if (maChaîne.length == 3) {
    return true;
  } else {
    alert(
      "Veuillez saisir trois caractères. " + maChaîne + " n'est pas valide.",
```

```
      );
      return false;
    }
  }
```

L'instruction `switch`

L'instruction `switch` permet à un programme d'évaluer une expression et d'effectuer des instructions en fonction des différents cas de figures correspondants aux différentes valeurs. Si un cas correspond au résultat de l'évaluation, le programme exécute l'instruction associée.

Voici un exemple utilisant une instruction `switch` :

JS

```
switch (expression) {
  case label_1:
    instructions_1
    [break;]
  case label_2:
    instructions_2
    [break;]
  ...
  default:
    instructions_par_defaut
    [break;]
}
```

JavaScript évalue l'instruction de commutation ci-dessus comme suit :

- Le programme recherche d'abord une clause `case` dont l'étiquette correspond à la valeur de l'expression, puis il transfère le contrôle à cette clause, en exécutant les instructions associées.
- Si aucune étiquette correspondante n'est trouvée, le programme recherche la clause optionnelle `default` :
  - Si une clause `default` est trouvée, le programme transfère le contrôle à cette clause, exécutant les déclarations associées.

- Si aucune clause `default` n'est trouvée, le programme reprend l'exécution à l'instruction qui suit la fin de `switch`.
- (Par convention, la clause `default` est écrite comme la dernière clause, mais il n'est pas nécessaire que ce soit le cas).

L'instruction break

L'instruction optionnelle `break`, éventuellement contenue pour chaque clause `case`, permet de ne pas exécuter les instructions pour les cas suivants. Si `break` n'est pas utilisé, le programme continuera son exécution avec les autres instructions contenues dans l'instruction `switch`.

EXEMPLE

Dans l'exemple suivant, si `fruit` vaut "Banane", le programme exécutera les instructions associées. Quand `break` est rencontré, le programme passe aux instructions décrites après `switch`. Ici, si `break` n'était pas présent, les instructions pour le cas "Cerise" auraient également été exécutées.

JS

```
switch (fruit) {
  case "Orange":
    console.log("Les oranges sont à 60 centimes le kilo.");
    break;
  case "Pomme":
    console.log("Les pommes sont à 32 centimes le kilo.");
    break;
  case "Banane":
    console.log("Les bananes sont à 48 centimes le kilo.");
    break;
  case "Cerise":
    console.log("Les cerises sont à 3€ le kilo.");
    break;
  case "Mangue":
    console.log("Les mangues sont à 50 centimes le kilo.");
    break;
  default:
    console.log("Désolé, nous n'avons pas de " + fruittype + ".");
}
console.log("Souhaitez-vous autre chose ?");
```

Les instructions pour gérer les exceptions

Il est possible de lever des exceptions avec l'instruction `throw` et de les gérer (les intercepter) avec des instructions `try...catch`.

- L'instruction throw
- L'instruction `try...catch`

Les types d'exception

En JavaScript, n'importe quel objet peut être signalé comme une exception. Cependant, afin de respecter certaines conventions et de bénéficier de certaines informations, on pourra utiliser les types destinés à cet effet :

- Les exceptions ECMAScript
- `DOMException` et `DOMError`

L'instruction `throw`

L'instruction `throw` est utilisée afin de signaler une exception. Lorsqu'on signale une exception, on définit une expression qui contient la valeur à renvoyer pour l'exception :

JS

```
throw expression;
```

Il est possible d'utiliser n'importe quelle expression, sans restriction de type. Le fragment de code qui suit illustre les différentes possibilités :

JS

```
throw "Erreur2"; //type String
throw 42; //type Number
throw true; //type Boolean
throw {
  toString: function () {
    return "je suis un objet !";
  },
};
```

> **Note :** Vous pouvez spécifier un objet lorsque vous lancez une exception. Vous pouvez alors faire référence aux propriétés de l'objet dans le bloc `catch`.

JS

```
// On crée le constructeur pour cet objet
function ExceptionUtilisateur(message) {
  this.message = message;
  this.name = "ExceptionUtilisateur";
}

// On surcharge la méthode toString pour afficher
// un message plus explicite (par exemple dans la console)
ExceptionUtilisateur.prototype.toString = function () {
  return this.name + ': "' + this.message + '"';
};

// On crée une instance pour ce type d'objet
// et on renvoie une exception avec cette instance
throw new ExceptionUtilisateur("La valeur fournie est trop élevée.");
```

L'instruction `try...catch`

L'instruction `try...catch` permet de définir un bloc d'instructions qu'on essaye (*try* en anglais) d'exécuter, ainsi qu'une ou plusieurs instructions à utiliser en cas d'erreur lorsqu'une exception se produit. Si une exception est signalée, l'instruction `try...catch` permettra de l' « attraper » (*catch* en anglais) et de définir ce qui se passe dans ce cas.

L'instruction `try...catch` se compose d'un bloc `try` qui contient une ou plusieurs instructions et blocs `catch` qui contiennent les instructions à exécuter lorsqu'une exception se produit dans le bloc `try`.

Autrement dit, dans la plupart des cas pour le programme, on veut que les instructions du bloc `try` se déroulent normalement et en cas de problème, on passe le contrôle au bloc `catch`. Si une instruction contenue dans le bloc `try` renvoie une exception, le contrôle sera immédiatement transféré au bloc `catch`. Si aucune exception n'est signalée au sein du bloc `try`, le bloc `catch` ne sera pas utilisé. Cette instruction peut comporter un bloc `finally` qui s'exécute après les blocs `try` et `catch` mais avant les instructions suivant l'instruction `try...catch`.

Dans l'exemple qui suit, on utilise une instruction `try...catch`. On définit une fonction qui prend un nombre et renvoie le nom du mois correspondant à ce nombre. Si la valeur fournie n'est pas comprise entre 1 et 12, on signale une exception avec la valeur `"NuméroMoisInvalide"`. Lorsque cette exception est gérée dans le bloc `catch`, la variable `nomMois` recevra la valeur `"inconnu"`.

JS

```
function getNomMois(numMois) {
  numMois = numMois - 1; // On décale de 1 car les indices du tableaux commencent à 0
  var mois = [
    "Janvier",
    "Février",
    "Mars",
    "Avril",
    "Mai",
    "Juin",
    "Juillet",
    "Août",
    "Septembre",
    "Octobre",
    "Novembre",
    "Décembre",
  ];
  if (mois[numMois] != null) {
    return mois[numMois];
  } else {
    throw "NuméroMoisInvalide"; // Ici on utilise l'instruction throw
  }
}

try {
  // les instructions à essayer si tout se passe bien
  nomMois = getNomMois(maVarMois); // La fonction peut renvoyer une exception
} catch (e) {
  nomMois = "inconnu";
  gestionErreurLog(e); // on gère l'erreur avec une fonction
}
```

Le bloc `catch`

Un bloc `catch` peut être utilisé afin de gérer les exceptions pouvant être générées par les instructions du bloc `try`.

JS

```
catch (ident) {
  statements
}
```

Le bloc `catch` définit un identifiant (`ident` dans le fragment de code précédent) qui contiendra la valeur passée par l'instruction `throw` . Cet identifiant peut être utilisé afin de récupérer des informations sur l'exception qui a été signalée.

JavaScript crée cet identifiant lorsque le contrôle passe au bloc `catch` . L'identifiant ne « vit » qu'à l'intérieur du bloc `catch` et une fois que l'exécution du bloc `catch` est terminée, l'identifiant n'est plus disponible.

Dans l'exemple suivant, le code renvoie une exception. Lorsque celle-ci est signalée, le contrôle passe au bloc `catch` .

JS

```
try {
  throw "monException"; // on génère une exception
} catch (e) {
  // les instructions utilisées pour gérer les exceptions
  enregistrerErreurs(e); // on passe l'objet représentant l'exception à une fonction utilisée pour gérer les erreurs
}
```

> **Note :** Quand on souhaite afficher des erreurs dans la console, on privilégiera `console.error()` plutôt que `console.log()` . En effet, cette première méthode est plus adaptée et indiquera plus d'informations.

Le bloc `finally`

Le bloc `finally` contient les instructions à exécuter après les blocs `try` et `catch` mais avant l'instruction suivant le `try...catch...finally` .

Le bloc `finally` est exécuté dans tous les cas, **qu'une exception ait été levée ou non**. Si une exception est signalée et qu'il n'y a pas de bloc `catch` pour la gérer, les instructions

du bloc `finally` seront tout de même exécutées.

Le bloc `finally` peut être utilisé afin de finir proprement l'exécution malgré une exception. On peut, par exemple, devoir libérer une ressource, ou fermer un flux, etc.

Dans l'exemple suivant, on écrit dans un fichier, si une exception se produit lors de l'écriture, on utilisera le bloc `finally` afin de bien fermer le flux vers le fichier avant la fin du script.

```
ouvrirFichier();
try {
  écrireFichier(données); // Une erreur peut se produire
} catch (e) {
  gérerException(e); // On gère le cas où on a une exception
} finally {
  fermerFichier(); // On n'oublie jamais de fermer le flux.
}
```

Si le bloc `finally` renvoie une valeur, cette valeur sera considérée comme la valeur de retour pour tout l'ensemble `try-catch-finally`, quelles que soient les instructions `return` éventuellement utilisées dans les blocs `try` et `catch` :

JS

```
function f() {
  try {
    console.log(0);
    throw "bug";
  } catch (e) {
    console.log(1);
    return true; // Cette instruction est bloquée jusqu'à la fin du bloc finally
    console.log(2); // Ne pourra jamais être exécuté
  } finally {
    console.log(3);
    return false; // On surcharge l'instruction "return" précédente
    console.log(4); // Ne pourra jamais être exécuté
  }
  // "return false" est exécuté

  console.log(5); // Ne pourra jamais être exécuté
```

```
}
f(); // affiche 0, 1, 3 puis renvoie false
```

Lorsqu'on surcharge les valeurs de retour avec le bloc `finally`, cela s'applique également aux exceptions qui sont levées (ou retransmises) au sein du bloc `catch` :

JS

```
function f() {
  try {
    throw "problème";
  } catch (e) {
    console.log('"problème" interne intercepté');
    throw e; // cette instruction est mise en attente
    // tant que le bloc finally n'est pas fini
  } finally {
    return false; // surcharge le "throw" précédent
  }
  // "return false" est exécuté à ce moment
}

try {
  f();
} catch (e) {
  // ce bloc n'est jamais utilisé car le throw
  // utilisé dans le bloc catch a été surchargé
  // par l'instruction return de finally
  console.log('"problème" externe intercepté');
}

// Sortie
// "problème" interne attrapé
```

Imbriquer des instructions `try...catch`

Il est possible d'imbriquer une ou plusieurs instructions `try...catch`.

Si un bloc `try` interne n'a *pas* de bloc `catch` correspondant :

1. il *doit* contenir un bloc `finally`, et

2. le bloc `try...catch` de l'instruction `catch` englobante est vérifié pour une correspondance.

Pour plus d'informations, voir nested try-blocks sur la page de référence `try...catch`.

Utiliser les objets d'erreur

En fonction du type d'erreur qui est créée, on pourra utiliser les propriétés `name` et `message` afin d'obtenir plus d'informations.

Généralement on a `name` qui fournit le type d'erreur rencontrée (ex : `DOMException` ou `Error`). La propriété `message`, quant à elle fournit un message descriptif de l'erreur (qu'on utilisera généralement lorsqu'on voudra convertir/afficher le texte correspondant à une erreur).

Si vous construisez des erreurs, vous pouvez utiliser le constructeur `Error` afin de disposer de ces propriétés.

Ainsi, on pourra avoir :

JS

```
function causerErreurs() {
  if (toutEstSourceDErreurs()) {
    throw (new Error('mon message'));
  } else {
    générerUneAutreErreur();
  }
}
....
try {
  causerErreurs();
} catch (e) {
  console.error(e.name);// affiche 'Error'
  console.error(e.message); // affiche 'mon message' ou un message d'erreur
JavaScript
}
```

This page was last modified on 3 août 2023 by MDN contributors.