# Programmation web
# JavaScript - Langage 5 - Programmation asynchrone

Vincent Padovani, PPS, IRIF

JavaScript s'exécute dans un environnement auquel il peut demander d'effectuer certaines opérations de manière *asynchrone* : une demande d'opération asynchrone n'est pas bloquante pour l'exécution du programme, mais sa complétion peut ne pas être immédiate. Il peut s'agir par exemple du chargement d'un fichier local ou distant, de la réception d'une réponse à l'envoi d'une requête HTTP, etc.

JavaScript étant un langage à fil d'exécution unique, une manière usuelle de récupérer le résultat d'une opération asynchrone est d'associer à sa demande une "fonction de rappel" (callback) qui sera exécutée à la complétion normale de l'opération. Il est courant d'associer à cette demande une seconde fonction de rappel qui sera, elle, exécuté en cas d'échec : on parle alors plutôt d'un couple formé d'un "gestionnaire d'echec" et d'un "gestionnaire de réussite" (success handler, failure handler).

Noter que, toujours à cause de l'unicité du fil d'exécution, l'exécution d'une fonction de rappel est elle-même une opération asynchrone : elle ne pourra avoir lieu que lorsque toutes les instructions de cet unique fil auront été complétées – une boucle infinie est par exemple bloquante pour cette exécution.

Ce chapitre présente un ensemble d'outils syntaxiques permettant de mettre en forme de telles demandes d'opérations asynchrones. En préliminaire, nous décrirons la syntaxe des lancements et captures d'exceptions.

## 1 Les exceptions

La syntaxe des exceptions en JavaScript est sans surprise : à l'absence de typage près, elle est similaire à celle de Java et obéit aux mêmes règles de comportement.

Le lanceur d'exception s'écrit `throw`, il peut être suivi d'une expression quelconque. En pratique, le constructeur prédéfini `Error` permet de construire un descripteur d'erreur encapsulant un message choisi par le programmeur, mais aussi d'autres informations utiles sur l'état du programme au moment du lancement de l'exception (nom du fichier, numéro de ligne, ...) :

```
throw new Error("erreur...");
```

Par défaut, une exception se propage à travers toute la pile d'appel en interrompant l'exécution de chaque fonction, puis celle du programme. La capture d'une exception s'effectue avec l'habituelle structure `try`/`catch`, éventuellement complétée par `finally` :

```
try {
  // exécuté jusqu'au bout, sauf si l'exécution atteint
  // directement ou indirectement un lancement d'exception
  // en throw ... : dans ce cas, passage immédiat à catch.
}
catch (e) {
  // exécuté si une exception est lancée dans le bloc try,
  // avec e désignant l'argument du throw ayant lancé cette
  // exception. la portée de e est locale.
}
finally {
  // optionnel, exécuté dans tous les cas après try/catch
}
```

# 2 Les promesses

Les promesses en JavaScript fournissent une syntaxe naturelle pour spécifier une suite d'opérations sur un ensemble de données lorsque chaque nouvelle étape nécessite les données produites à l'étape précédente, et lorsque une ou plusieurs de ces étapes nécessitent des opérations asynchrones. Chaque étape ne peut bien sûr démarrer son traitement que lorsque ses données sont prêtes, c'est-à-dire lorsque les étapes précédentes ont toutes été complétées. Il est également possible que l'une de ces étapes échoue (ressource indisponible ou corrompue, par exemple), et les promesses permettent aussi de gérer ces échecs.

## 2.1 Construction d'une chaîne de promesses

Une *chaîne de promesses* est un assemblage d'objets donc chaque élément, une *promesse*, est destiné à effectuer une des étapes d'un traitement asynchrone.. Elle se construit à partir d'un premier objet initialisé par le constructeur `Promise`, en fournissant à ce constructeur un argument fonctionnel appelé *exécutant* de la promesse créée. L'exécutant doit être une fonction à deux paramètres eux-mêmes fonctionnels, traditionnellement appelés `resolve` et `reject`.

```
let p = new Promise((resolve, reject) => { /* corps de l'exécutant */ });
```

Les promesses suivantes sont ajoutés à la chaîne à l'aide de la méthode `then` de `Promise.prototype`. Cette méthode a deux paramètres fonctionnels pouvant recevoir chacun une fonction unaire. En retour, cette méthode renvoie une *nouvelle* promesse, c'est-à-dire un objet héritant lui aussi d'une méthode `then`, ce qui permet d'ajouter de la même manière les éléments suivants de la chaîne :

```
let p = new Promise((resolve, reject) => { /* ...  */ });
let q = p.then((data) => { /* .... */ }, (err) => { /* ... */ })
let r = q.then((data) => { /* .... */ }, (err) => { /* ... */ })
// etc.
```

L'usage de variables intermédiaires dans cette construction n'est pas indispensable, on peut aussi écrire :

```
let p = new Promise((resolve, reject) => {
  // ...
}).then((data) => { /* .... */ }, (err) => { /* ... */ })
  .then((data) => { /* .... */ }, (err) => { /* ... */ })
   // etc.
```

**Gestionnaires de réussite et d'échec.** Dans chacune des promesses résultant d'une invocation de `then`, la première fonction en `data` argument de la méthode est appelé *gestionnaire de réussite* de la promesse construite, la seconde fonction en `err` est son *gestionnaire d"echec*.

Le gestionnaire d'échec est optionnel dans cette invocation, ce qui permet d'ajouter à une chaîne une promesse munie seulement un gestionnaire de réussite. Une promesse peut être aussi seulement munie d'un gestionnaire d'échec, en donnant comme premier argument à `then` la valeur `null`. La méthode `catch` de `Promise.prototype` fournit un raccourci d'écriture pour cet ajout – les deux invocations ci-dessous sont équivalentes :

```
q.catch((err) => { /* ... */ }); // q.then(null, (err) => { /* ... */ });
```

**Prochains gestionnaires de réussite et d'échecs.** Etant donné un élément d'une chaîne de promesses – l'exécutant, un gestionnaire de promesse – ce que nous appellerons *prochain gestionnaire* de réussite (resp. d'échec) relativement à cet élément est le gestionnaire de réussite (resp. d'échec) de la première promesse muni d'un tel gestionnaire dans la suite des promesses chaînées à la promesse de cet élément.

**Forme standard d'une chaîne.** Même s'il est possible d'invoquer `then` en lui fournissant deux gestionnaires, ce n'est pas la manière dont cette méthode est utilisée en pratique. La forme suivante est presque toujours celle utilisée en programmation réelle – une suite d'ajouts de promesses munies seulement d'un gestionnaire de réussite, un ajout final d'une promesse munie seulement d'un gestionnaire d'échec :

```
// forme standard d'une chaîne promesses
let p = new Promise((resolve, reject) => {
   // ...
}).then((data) => { /* ... */ })
   // ...
  .then((data) => { /* ... */ })
  .catch((err) => { /* ... */ });
```

La manière exacte dont une telle chaîne est exécutée sera décrite en détail à la section suivante, mais on peut en donner déjà une description intuitive. Un traitement séquentiel effectué par une chaîne standard donnera a priori à ses différents éléments les responsabilités suivantes :

- Le rôle de l'exécutant (l'argument de `Promise`) sera de démarrer le traitement, par exemple en chargeant ou en initialisant un ensemble de ressources et, en l'absence d'erreur, de transmettre les données collectés au premier gestionnaire de réussite.
- Le rôle de chaque gestionnaire de réussite autre que le dernier sera d'effectuer une étape supplémentaire du traitement (soumettre les données à un parser, les décompresser, les décrypter, etc.) et, en l'absence d'erreur, de transmettre les données modifiées au gestionnaire de réussite suivant.
- Le rôle du dernier gestionnaire de réussite sera d'utiliser de manière effective le résultat du traitement (le présenter à l'utilisateur, l'insérer dans une page web, l'ajouter à une base de données, etc.).
- Le rôle du gestionnaire d'échec sera de gérer les erreurs pouvant se produire à l'une ou l'autre de ces étapes.

## 2.2 Exécution d'une chaîne de promesses

Le code de l'exécutant ou celui des gestionnaires d'une chaîne s'exécute toujours de manière asynchrone, c'est-à-dire après exécution de toutes les instructions du fil d'exécution normal du programme (appels de fonctions, boucles, etc.).

Un exécutant ou un gestionnaire atteint pendant l'exécution d'une chaîne peut effectuer une et une seule fois l'une des actions suivants. Tant qu'il n'a pas effectué l'une d'elles sa promesse associée est dite *en attente* (pending).

1. Demander qu'après son exécution complète soit effectué un appel du prochain gestionnaire de réussite, sur l'argument `value` de son choix, pourvu qu'il ne s'agisse pas d'une promesse. Syntaxiquement, cette action s'écrit :
   - dans un exécutant : `resolve(value);`
   - dans un gestionnaire : `return value;`

   Le promesse associée à l'élément ayant effectué cette action est dite *tenue* (fullfilled) et *associée à la valeur* `value`.
2. Seulement s'il est exécutant, demander qu'après son exécution complète soit effectué un appel du prochain gestionnaire d'échec, sur l'argument `err` de son choix. La syntaxe de cette action est : `reject(err);`. La promesse associée à l'élément ayant effectué cette action est dite *rompue* (rejected) *en raison de l'erreur* `err`.
3. Lancer une exception par une instruction de la forme `throw err;` Ce lancement interrompt l'exécution de l'élément, et est suivi d'un appel du prochain gestionnaire d'échec avec pour argument `err`. Comme précédemment, la promesse associée à l'élément ayant effectué cette action est dite rompue en raison de l'erreur `err`.
4. Déléguer le choix d'une action à l'exécutant ou au dernier gestionnaire atteint d'une autre promesse, par exemple une promesse produite par un appel de fonction ou construite à la volée. Cette action s'écrit comme la première, mais en remplaçant `value` par cette autre promesse.

Dans les trois premières sortes d'actions, en l'absence d'un prochain gestionnaire de la forme souhaitée, l'appel demandé ne se fera qu'au premier ajout à la chaîne d'une promesse munie du gestionnaire approprié. La promesse d'un exécutant terminant son exécution sans avoir effectué aucune de ces actions reste "en attente" (pending). Dans le cas d'un exécutant en revanche, la promesse passe dans l'état "tenue" (fullfilled), mais elle est associée à la valeur `undefined`.

**Exemples.** Voici un exemple d'une suite d'actions de la première forme effectuées par un exécutant et plusieurs occurences d'un gestionnaire de réussite. Noter que le premier message (`"REACHED"`) est effectivement affiché, puisque la première occurrence de `success` ne sera appelée qu'après terminaison de l'exécutant. Noter également la valeur associée à `p` après les deux exécutions de `success` : `44 === ((42 + 1) + 1)`.

```
function success(value) {
  console.log("(SUCCESS) " + value);
  return value + 1;
}

function failure(err) {
  console.log("(ERROR) " + err);
}

let p = new Promise((resolve, reject) => {
  resolve(42);
  console.log("(REACHED)");
}).then(success)
  .then(success)
  .catch(failure); // => (REACHED)
                   // => (SUCCESS) 42
                   // => (SUCCESS) 43
                   // => Promise { <state>: "fulfilled", <value>: 44 }

p.then(success);   // => (SUCCESS) 44
                   // => Promise { <state>: "fulfilled", <value>: 45 }
p.then(success)
 .then(success);   // => (SUCCESS) 44
                   // => (SUCCESS) 45
                   // => Promise { <state>: "fulfilled", <value>: 46 }

p;                 // => Promise { <state>: "fulfilled", <value>: 44 }
```

Autre exemple avec cette fois une action de la seconde forme, une rupture de la promesse de l'exécutant. Noter que seule la promesse `q` est rompue : dans la chaîne construite après sa déclaration, le gestionnaire `failure` se contente de terminer son exécution sans aucune action. Sa promesse associée est donc tenue, mais associée à une valeur indéfinie.

```
let q = new Promise((resolve, reject) => {
  reject(new Error("erreur"));
  console.log("(REACHED)");
});              // => (REACHED)
                 // => Promise {<state>: "rejected", <reason>: Error}
q.then(success)
 .then(success)
 .catch(failure);// => (ERROR) Error: erreur
                 // => Promise {<state>: "fulfilled", <value>: undefined}
```

Autre exemple, une action de la troisième forme, le lancement d'un exception – ici, dans l'exécutant, mais qui pourrait être dans un gestionnaire quelconque. La différence avec l'exemple précédent est l'interruption immédiate de l'exécutant de `r` : l'instruction d'affichage n'est plus atteinte.

```
let r = new Promise((resolve, reject) => {
  throw new Error("erreur");
  console.log("(UNREACHED)");
});              //  => Promise {<state>: "rejected", <reason>: Error}
r.then(success)
 .then(success)
 .catch(failure);// => (ERROR) Error: erreur
                 // => Promise {<state>: "fulfilled", <value>: undefined}
```

Deux exemple d'actions de la quatrième forme, la délégation à une autre promesse du choix de l'action d'un exécutant – cet exemple est simpliste et ne fait qu'illustrer le principe de cette délégation, la promesse déléguée pourrait elle-même déléguer ce choix à une autre promesse, qui pourrait le déléguer à une autre, etc.

```
var p0 = new Promise((resolve, reject) => {
  console.log("P0");
  let p1 =  new Promise((resolve, reject) => {
    console.log("P1");
    resolve(42);
  })
  resolve(p1);      // choix de l'action délégué à p1
}).then(success)
  .then(success)
  .catch(failure)   // => P0
                    // => P1
                    // (SUCCESS) 42
                    // (SUCCESS) 43

var q0 = new Promise((resolve, reject) => {
  console.log("Q0");
  let q1 =  new Promise((resolve, reject) => {
    console.log("Q1");
    reject(new Error "erreur");
  })
  resolve(q1);      // choix de l'action délégué à q1
}).then(success)
  .then(success)
  .catch(failure)   // => Q0
                    // => Q1
                    // => (ERROR) Error: erreur
```

Deux autres exemples du principe de délégation, cette fois dans des gestionnaires :

```
let r0 = new Promise((resolve, reject) => {
  console.log("R0");
  resolve(42);
}).then(value => {
     console.log("R1");
     // choix de l'action délégué à cette promesse :
     return new Promise ((resolve, reject) => {
       console.log("R2");
       resolve(value + 1);
     });
}).then(success)
  .then(success)
  .catch(failure);  // R0
                    // R1
                    // R2
                    // (SUCCESS) 43
                    // (SUCCESS) 44

let s0 = new Promise((resolve, reject) => {
  console.log("S0");
  resolve(42);
}).then(value => {
     console.log("S1");
     // choix de l'action délégué à cette promesse :
     return new Promise ((resolve, reject) => {
       console.log("S2");
       reject(new Error("erreur"));
     });
}).then(success)
  .then(success)
  .catch(failure);  // S0
                    // S1
                    // S2
                    // => (ERROR) Error: erreur
```

**Remarques.** Deux méthodes de `Promise` fournissent les raccourcis d'écriture suivants :

```
// let p = new Promise((resolve, reject) => resolve(42));
let p = new Promise.resolve(42);
// let q = new Promise((resolve, reject) => reject(new Error("err")));
let q = new Promise.reject(new Error("err"));
```

Par ailleurs, certains auteurs qualifient d'*acquittées* (settled) les promesses dans l'état "tenue" ou "rompue" (fulfilled, rejected), et de *résolues, mais non encore acquitées* (resolved but not fulfilled yet) celles non encore acquitées après une délégation. Il existe d'autres terminologies plus exotiques encore pour décrire les états d'une promesse, mais elles n'apportent qu'un éclairage douteux sur cette notion, déjà suffisamment difficile.

# 3 `async` et `await`

## 3.1 `async`

Le mot-clef `async` offre un raccourci d'écriture pour la définition de fonctions construisant de nouvelles promesses à partir d'un ensemble d'arguments. Un appel d'une fonction `async` s'effectue de manière synchrone, mais une promesse renvoyée par cette fonction s'exécutera, comme toute autre promesse, de manière asynchrone. Le corps de la fonction `async` peut être vu comme décrivant le traitement asynchrone effectué par cette promesse. Son exécution peut se terminer de deux manières :

1. En atteignant une instruction en `return`. Dans ce cas, la promesse construite atteindra comme d'habitude l'état "tenu" (fullfilled) en étant associé à la valeur de l'expression qui suit ce `return`.
2. En atteignant un lancement d'exception en `throw`, non capturée par une autre partie du corps de la fonction. Dans ce cas, la promesse atteindra l'état "rompu" (rejected) en raison de l'erreur qui suit le lanceur de cette exception.

Exemple :

```
async function divide(x, y) {
  if (y === 0) {
    throw new Error("Division par zero error");
  }
  return x / y;
}
divide;       // => async function divide(x, y)
divide(42, 2) // => Promise { <state>: "fulfilled", <value>: 21 }
divide(42, 0) // => Promise { <state>: "rejected", <reason>: Error }
```

Le code précédent est donc opératoirement équivalent à celui-ci, sans le mot-clef `async` :

```
function divide(x, y) {
  function body(x, y) {
    if (y === 0) {
        throw new Error("Division par zero error");
    }
    return x / y;
  }
  return new Promise((resolve, reject) => {
    try {
      resolve(body(x, y))
    }
    catch(err) {
      reject(err);
    }
  });
}
```

## 3.2 `await`

Le mot-clef `await` ne peut être utilisé que dans le corps d'une fonction `async`. Il doit toujours être suivi d'une promesse, et sert à construire des expressions s'évaluant de la manière suivante :

1. La valeur de l'expression ne pourra être déterminée que lorsque la promesse qui suit `await` aura quitté l'état "en attente" (pending). Tant que cette promesse n'aura pas atteint un état différent, le corps de la fonction `async` ne reprendra pas son exécution.

   Dans une suite d'instructions de la forme suivante, par exemple, où `p`, `q` et `r` sont des promesses, `a` ne prendra une valeur que lorsque `p` aura quitté sont état "pending", `b` ne prendra une valeur que lorsque `a` aura pris une valeur et `q` aura quitté sont état "pending", etc.

   ```
   let a = await p;
   let b = await q;
   let c = await r;
   ```

2. Si la promesse qui suit un `await` atteint l'état "tenue" (fulfilled), la valeur de l'expression est celle qui est alors associée à la promesse. Dans le code ci-dessus, si `p` atteint l'état "tenue" en étant associé à la valeur `42`, la valeur prise par `a` est `42`.
3. Si la promesse qui suit `await` atteint l'état "rompue" (rejected) en raison d'une erreur, l'exécution de la fonction `async` est interrompue, et la promesse globale construite par cette fonction passe dans l'état "rompue" pour la même raison.

La fonction du mot-clef `await` est donc de forcer la séquentialité de l'exécution du corps d'une fonction `async`, même lorsque des expressions mettent en jeu des promesses, qui doivent être tenues avant que les instructions suivantes ne puissent être exécutées. Les mécanismes sous-jacents à cette séquentialité forcée mettent évidemment en jeu des chaînes de promesses, mais la syntaxe rend ces détails transparents pour le programmeur. Un exemple :

```
function delayedValue(delay, value) {
  return new Promise((resolve, reject) => {
    console.log("start : " + value);
    function end() {
      console.log("end   : " + ++value);
      resolve(value);
    }
    setTimeout(end, delay);
  });
}
async function delayedComputation(start) {
  let a = await delayedValue(1000, start);
  let b = await delayedValue(1000, a);
  let c = await delayedValue(1000, b);
  return c;
}
```

La fonction prédéfinie `setTimeOut` permet de demander l'appel d'une fonction quelconque au bout d'un certain nombre de millisecondes. Elle est ici utilisée dans la fonction `delayedValue` pour construire une nouvelle promesse qui ne devient tenue et associée à une valeur qu'au bout d'un certain délai.

La fonction `async delayedComputation` construit une promesse à partir d'une valeur de départ, qui est incrémentée trois fois, de manière indirecte, via une promesse construite avec `delayedValue` : à partir d'une valeur donnée, cette promesse passe dans l'état "tenue" (fullfilled) en étant associée au successeur de cette valeur, mais seulement une seconde après le lancement de son exécutant.

Séquentiellement : `a` prend la valeur `43`, `b` prend la valeur `44`, `c` prend la valeur `45`, puis la promesse construite par la fonction `async` pass dans l'état "tenue" (fullfilled) en étant associée à la valeur qui suit `return`, c'est-à-dire celle de `c` :

```
let p = delayedComputation(42);

// => start 42
// => end   43     // (une seconde plus tard)
// => start 43
// => end   44     // (une seconde plus tard)
// => start 44
// => end   45     // (une seconde plus tard)

p;                 // Promise { <state>: "fulfilled", <value>: 45 }
```

# 4 Itérateurs asynchrones

## 4.1 Objets itérables de manière asynchrone

Rappelons les définitions vues au Chapitre sur les itérations :

- Un résultat d'itération est un objet muni d'une propriété `value` ou une propriété `done`. Ce résultat est *valide* si sa propriété `value` est définie et sa propriété `done` non égale à `false`, et une *fin d'itération* si sa propriété `done` égale à `false`.
- Un *itérateur* est un objet muni d'une méthode `next` renvoyant un résultat d'itération. Les résultats renvoyés par `next` doivent être valides jusqu'à ce que cette méthode renvoie pour la première fois une fin d'itération.
- Un objet est *itérable* s'il possède une méthode `[Symbol.iterator]` renvoyant un itérateur.

Un *itérateur asynchrone* est un objet muni d'une méthode `next` renvoyant non pas un résultat d'itération, mais une *promesse* qui, si elle est tenue, sera associée à un résultat d'itération.

Les itérateurs asynchrones s'utilisent par exemple pour manipuler une à une les données d'un flux, lorsque la production de ces données est elle-même une opération asynchrone. Le passage par cet intermédiaire d'une promesse permet de gérer le cas ou la production du résultat nécessite eventuellement un certain délai.

Tout objet peut être muni d'un itérateur asychrone : il suffit de le munir d'une méthode `[Symbol.asyncIterator]` renvoyant un tel itérateur.

## 4.2 Boucles `for`/`await`/`in`

Rappelons qu'une boucle `for`/`in` permet sur un objet énumérable d'examiner une à une les valeurs encapsulées dans les résultat valides renvoyées par la méthode `next` de son itérateur.

De la même manière, dans une fonction `async`, une boucle `for`/`await`/`in` permet d"examiner une à une les valeurs encapsulées dans les résultat valides *associés de manière asynchrone aux promesses* renvoyées par la méthode `next` de son itérateur. Les deux formes suivantes sont équivalentes, et sont à comparer avec les deux formes équivalentes présentées dans le chapitre sur les itérateurs – les seules différences sont les ajouts du mot-clef `await` :

```
let asyncIterable = ...;
let asyncIterator = asyncIterable[Symbol.asyncIterator]();
for (let res = await asyncIterator.next(); !res.done;
         res = await asyncIterator.next()) {
  let data = res.value;
  // examiner data
}
```

```
let asyncIterable = ...;
for await (let data in asyncIterable) {
  // examiner data
}
```

**Exemple.** Nous allons dans cet exemple construire la simulation d'un flux de données, émettant une fois par seconde un élément de la suite d'entiers `0`, `1`, `2`, ... jusqu'à une certaine borne. Ce flux sera muni d'un itérateur asynchrone, permettant de récupérer l'entier suivant émis par le flux, par l'intermédiaire d'une promesse.

Les premiers éléments de cette simulation sont deux fonctions utilitaires construisant respectivement une promesse qui sera associée au bout d'un certain délai à un résulat d'itération valide, et une promesse immédiatement associée à une fin d'itération :

```
function delayedValidResult(data, delay) {
  return new Promise (resolve => {
    setTimeout(() => resolve({value : data}), delay);
  });
}
function finalResult () {
  return new Promise (resolve => {done : true});
}
```

Le second est un constructeur permettant de construire un flux. Chaque nouveau flux est muni d'un itérateur asynchrone : sa méthode `next` renvoie à chaque appel une nouvelle promesse associée à la valeur `{value : 0}` au bout d'un certain délai, puis à `{value : 1}` au bout du même délai, etc., puis à la valeur `{done : true}` :

```
function Stream (nbr, delay) {
  this.nbr = nbr;
  this[Symbol.asyncIterator] = function () {
    let data = 0;
    return {
      next : function () {
        return data < nbr ?
        delayedValidResult(data++, delay) :
        finalResult();
      }
    };
  };
}
```

Voici à présent un exemple de fonction `async` affichant les éléments du flux construit à la volée, une fois toutes les secondes :

```
async function streamReader(nbr, ms) {
  let stream = new Stream(nbr, ms);
  for await (let data of stream) {
    console.log(data);
  }
}
streamReader(5, 1000);  // => 0, 1, 2, 3, 4 (toutes les secondes)
```

Autre exemple de fonction `async`, collectant cette fois toutes les données du flux avant de les afficher :

```
async function streamCollecter(nbr, ms) {
  let stream = new Stream(nbr, ms);
  let dataArray = [];
  for await (let data of stream) {
    dataArray.push(data);   // ajouter data à la fin de dataArray
  }
  console.log(dataArray);
}
streamCollecter(5, 1000);
// => Array(5) [ 0, 1, 2, 3, 4 ] (au bout de 5 secondes)
```

## 4.3 Itérateurs et méthodes en `async`

Une méthode n'étant rien de plus qu'une propriété à valeur fonctionnelle, elle peut être déclarée comme `async`. Les promesses renvoyée par un `next` peuvent donc aussi être renvoyées de manière directe par un `next` en `async` à l'aide de simples `return` suivis de réponses. Voici une seconde version de l'exemple précédent, où une fonction auxiliaire se contente de construire une promesse tenue au bout d'un certain délai : un `await` sur cette promesse permettra de simuler l'attente de ce délai.

```
function delay(ms) {
  // retour d'une promesse tenue au bout de ms millisecondes,
  // associée à une valeur indéfinie.
  return new Promise (resolve => {
    setTimeout(() => resolve(), ms);
  });
}

function Stream (nbr, ms) {
  this.nbr = nbr;
  this[Symbol.asyncIterator] = function () {
    let data = 0;
    return {
      next : async function () {    // ou : async next() { ... }
        if (data >= nbr) {
          // retour implicite d'une promesse, à cause de async
          return {done : true};
        }
        let d = await delay(ms);
        // retour implicite d'une promesse, atteint seulement au
        // bout de ms millisecondes à cause du await précédent
        return {value : data++};
      }
    };
  };
}
```

## 4.4 Générateurs asynchrones

Une fonction génératrice peut être déclaré en `async`. Elle renvoie dans ce cas une générateur asynchrone, dont la première invocation de la méthode `next` entraîne l'exécution du corps de cette fonction génératrice comme un générateur ordinaire, mais de manière asynchrone : les valeurs qui suivent les instructions de production en `yield` seront encore encapsulées dans un résultat valide, mais ce résultat ne sera disponible que lorsqu'il deviendra associée à la promesse renvoyée par `next`.

Un exemple de fonction génératrice, avec la fonction `delay` précédente :

```
async function* stream(nbr, ms) {
  for (let i = 0; i < nbr; i++) {
    let d = await delay(ms);
    // instruction de production, atteinte au bout de ms millisecondes :
    yield i;
  }
}
```

Une autre version de la fonction `streamReader` invoquant cette fonction génératrice :

```
async function streamReader(nbr, ms) {
  let st = stream(nbr, ms);
  for await (let data of st) {
    console.log(data);
  }
}

streamReader(5, 1000); // 0 1 2 3 4 5 (toutes les secondes).
```