Pontificia Universidad Católica de Chile
Departamento de Ciencia de la Computación

IIC3585 Diseño Avanzado de Aplicaciones Web ∼ Segundo Semestre 2016

Iterators and Generators

Javascript ES6

Sebastián Soto Rojas (spsoto@uc.cl)

22 de Agosto, 2016

Iterators

Motivación

- Iterar sobre un **conjunto** de datos es una operación clásica en cualquier lenguaje de programación.
- En JavaScript se presenta el problema de que existen muchos tipos de conjuntos y distintas implementaciones de cómo consumir los datos de dichos conjuntos.
- En ES5, para hacer uso de métodos como `forEach`, `map`, `filter`, `reverse`, etc. se requiere hacer una conversión previa a Array o una colección similar o bien la implementación propia de los métodos.

Motivación

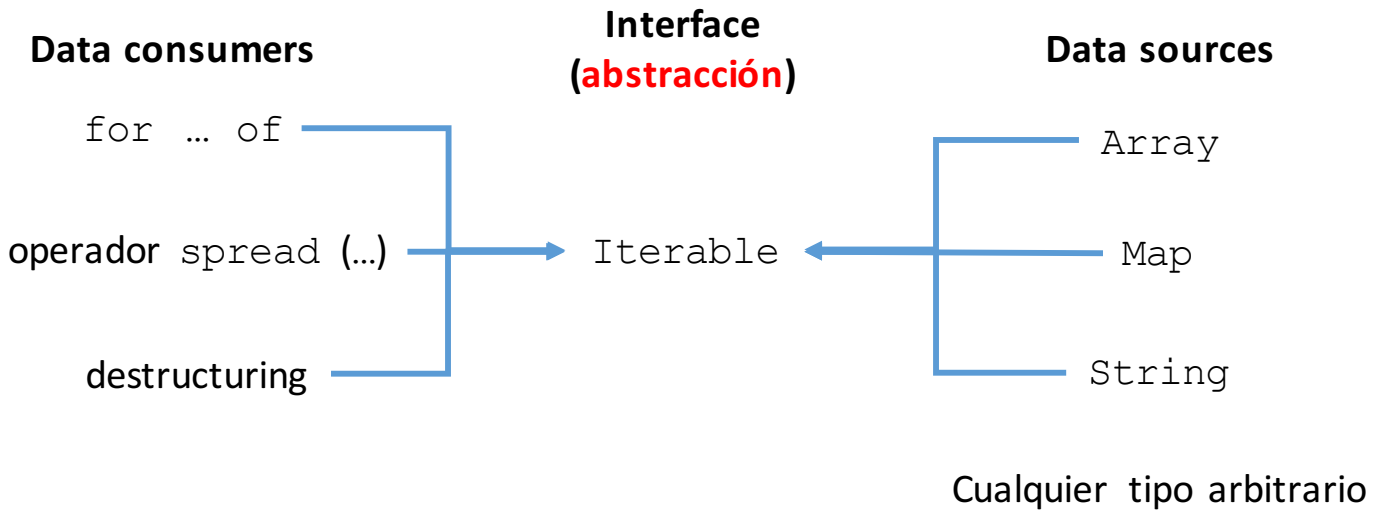
- Iterar sobre un **conjunto** de datos es una operación clásica en cualquier lenguaje de programación.
- En JavaScript se presenta el problema de que existen muchos tipos de conjuntos y distintas implementaciones de cómo consumir los datos de dichos conjuntos.
- En ES5, para hacer uso de métodos como `forEach`, `map`, `filter`, `reverse`, etc. se requiere hacer una conversión previa a Array o una colección similar o bien la implementación propia de los métodos.

Motivación

- Iterar sobre un **conjunto** de datos es una operación clásica en cualquier lenguaje de programación.
- En JavaScript se presenta el problema de que existen muchos tipos de conjuntos y distintas implementaciones de cómo consumir los datos de dichos conjuntos.
- En ES5, para hacer uso de métodos como `forEach`, `map`, `filter`, `reverse`, etc. se requiere hacer una conversión previa a Array o una colección similar o bien la implementación propia de los métodos.

Motivación

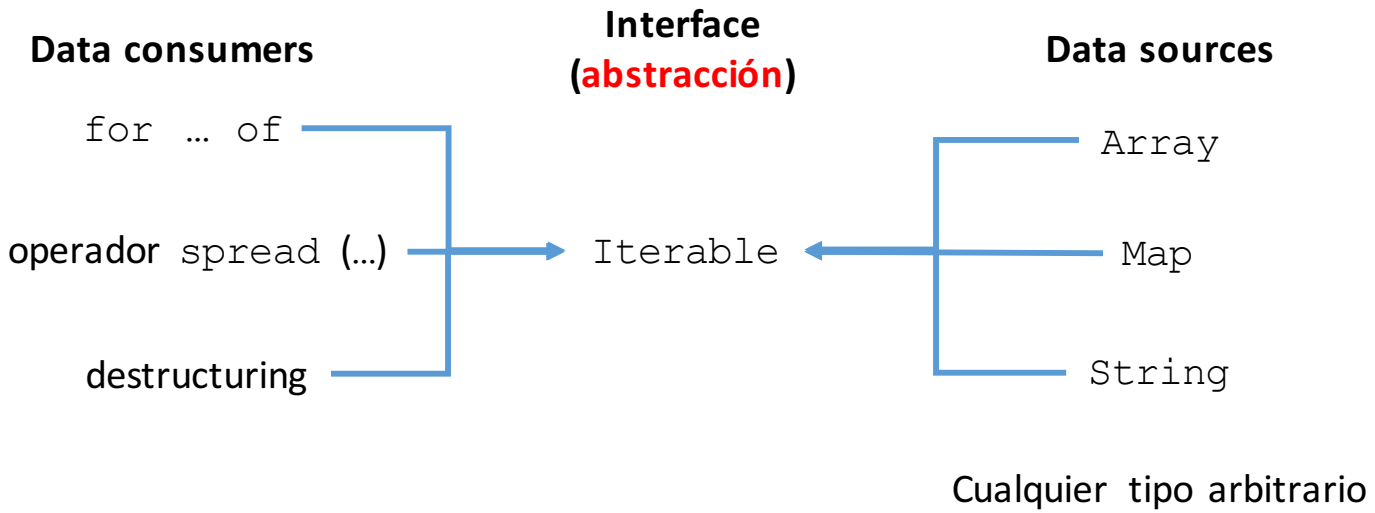
- Es decir, existen **Data Consumers** y **Data Sources**. Los Iterators de ES6 entregan una **interface** entre ambos.



- Es decir:
  - **Sources:** son *iterables* si tienen un método denominador `Symbol.iterator` que devuelve el iterador.
  - **Consumers:** usan el iterador para recibir valores.

Motivación

- Es decir, existen **Data Consumers** y **Data Sources**. Los Iterators de ES6 entregan una **interface** entre ambos.



- Es decir:
  - **Sources:** son *iterables* si tienen un método denominador `Symbol.iterator` que devuelve el iterador.
  - **Consumers:** usan el iterador para recibir valores.

Estructura y semántica básica

Cualquier objeto es un **iterador** si contiene entre sus atributos el método `next()`, el cual debe devolver un objeto con las siguiente semántica:

- `done` (boolean): `true` si se terminó de recorrer la secuencia, `false` o sin especificar en caso contrario.
- `value`: cualquier valor devuelto por el iterador. Puede omitirse si `done` es `true`.

Estructura y semántica básica

Cualquier objeto es un **iterador** si contiene entre sus atributos el método `next()`, el cual debe devolver un objeto con las siguiente semántica:

- `done` (boolean): `true` si se terminó de recorrer la secuencia, `false` o sin especificar en caso contrario.
- `value`: cualquier valor devuelto por el iterador. Puede omitirse si `done` es `true`.

Estructura y semántica básica

Cualquier objeto es un **iterador** si contiene entre sus atributos el método `next()`, el cual debe devolver un objeto con las siguiente semántica:

- `done` (boolean): `true` si se terminó de recorrer la secuencia, `false` o sin especificar en caso contrario.
- `value`: cualquier valor devuelto por el iterador. Puede omitirse si `done` es `true`.

Estructura y semántica básica

Código con la macroestructura:

```
var myIterator = function(){
  /* contenidos del metodo */

  return {
    next: function(){
      return {
        done: /* condicion de termino */,
        value: /* valor regresado */
      }
    }
    /* ... otros metodos opcionales (return, throw) */
  };
};
```

Objetos iterables: iterador por defecto

Cualquier objeto se vuelve iterable si implementa un método iterador, teniendo una propiedad llamada `Symbol.iterator` con dicho método:

```
var MyObject = function(){
  /* contenidos del objeto... */

  return {
    [Symbol.iterator]: function(){
      return this; /* "this" referencia al objeto */
    }
    next: function(){
      /* contenido del iterador */
    }
  }
}
```

Ejemplos

Los ejemplos pueden verse ejecutándolos vía Node:

- `ejemplo1.js`: Ejemplo básico de funcionamiento.
- `ejemplo2.js`: Ejemplo de un objeto iterable.
- `ejemplo3.js`: Forma de observar los iteradores de objetos nativos.
- `consumers.js`: Formas de consumir un iterable.

Ejemplos

Los ejemplos pueden verse ejecutándolos vía Node:

- `ejemplo1.js`: Ejemplo básico de funcionamiento.
- `ejemplo2.js`: Ejemplo de un objeto iterable.
- `ejemplo3.js`: Forma de observar los iteradores de objetos nativos.
- `consumers.js`: Formas de consumir un iterable.

Ejemplos

Los ejemplos pueden verse ejecutándolos vía Node:

- `ejemplo1.js`: Ejemplo básico de funcionamiento.
- `ejemplo2.js`: Ejemplo de un objeto iterable.
- `ejemplo3.js`: Forma de observar los iteradores de objetos nativos.
- `consumers.js`: Formas de consumir un iterable.

Ejemplos

Los ejemplos pueden verse ejecutándolos vía Node:

- `ejemplo1.js`: Ejemplo básico de funcionamiento.
- `ejemplo2.js`: Ejemplo de un objeto iterable.
- `ejemplo3.js`: Forma de observar los iteradores de objetos nativos.
- `consumers.js`: Formas de consumir un iterable.

Últimos comentarios

- `Promise.all` y `Promise.race` aceptan iterables.
- Puede agregarse método `return()` para controlar el estado de un término inesperado.
- Puede agregarse método `throw()` para interacción con generadores.
- Pueden extenderse al infinito. Basta especificar siempre un valor y no especificar `done`.

Últimos comentarios

- `Promise.all` y `Promise.race` aceptan iterables.
- Puede agregarse método `return()` para controlar el estado de un término inesperado.
- Puede agregarse método `throw()` para interacción con generadores.
- Pueden extenderse al infinito. Basta especificar siempre un valor y no especificar `done`.

Últimos comentarios

- `Promise.all` y `Promise.race` aceptan iterables.
- Puede agregarse método `return()` para controlar el estado de un término inesperado.
- Puede agregarse método `throw()` para interacción con generadores.
- Pueden extenderse al infinito. Basta especificar siempre un valor y no especificar `done`.

Últimos comentarios

- `Promise.all` y `Promise.race` aceptan iterables.
- Puede agregarse método `return()` para controlar el estado de un término inesperado.
- Puede agregarse método `throw()` para interacción con generadores.
- Pueden extenderse al infinito. Basta especificar siempre un valor y no especificar `done`.

Generators

Motivación

- La semántica anterior entrega una ventaja en la interacción de tipos de datos, **pero no en la facilidad de lectura**.
- En JavaScript asíncrono resulta ideal disponer de funciones cuya ejecución pueda ser pausada.
- Se crean los generadores para evitar todo el trabajo de definir iteradores, y controlar de forma automática las variables de estado (e.g. `done`).
- Son una herramienta poderosa en JavaScript, dada su naturaleza asíncrona.

Motivación

- La semántica anterior entrega una ventaja en la interacción de tipos de datos, **pero no en la facilidad de lectura**.
- En JavaScript asíncrono resulta ideal disponer de funciones cuya ejecución pueda ser pausada.
- Se crean los generadores para evitar todo el trabajo de definir iteradores, y controlar de forma automática las variables de estado (e.g. `done`).
- Son una herramienta poderosa en JavaScript, dada su naturaleza asíncrona.

Motivación

- La semántica anterior entrega una ventaja en la interacción de tipos de datos, **pero no en la facilidad de lectura**.
- En JavaScript asíncrono resulta ideal disponer de funciones cuya ejecución pueda ser pausada.
- Se crean los generadores para evitar todo el trabajo de definir iteradores, y controlar de forma automática las variables de estado (e.g. `done`).
- Son una herramienta poderosa en JavaScript, dada su naturaleza asíncrona.

Motivación

- La semántica anterior entrega una ventaja en la interacción de tipos de datos, **pero no en la facilidad de lectura**.
- En JavaScript asíncrono resulta ideal disponer de funciones cuya ejecución pueda ser pausada.
- Se crean los generadores para evitar todo el trabajo de definir iteradores, y controlar de forma automática las variables de estado (e.g. `done`).
- Son una herramienta poderosa en JavaScript, dada su naturaleza asíncrona.

Estructura y semántica

- Los generadores son **métodos** (¡no objetos!) definidos mediante la declaración `function*`.
- Siguen la estructura de cualquier otra función, pero:
  (a) Al llamarlos, se devuelve un objeto iterable.
  (b) Al ejecutar `next()` se ejecuta el cuerpo de la función.
  (c) Se ejecuta hasta llegar a la primera declaración `yield` o `return`.
- Aparece la declaración `yield`, la cual entrega el par `{done: ..., value: ...}` y pausa la ejecución hasta la próxima vez que se llama a `next()`.

Estructura y semántica

- Los generadores son **métodos** (¡no objetos!) definidos mediante la declaración `function*`.
- Siguen la estructura de cualquier otra función, pero:
  (a) Al llamarlos, se devuelve un objeto iterable.
  (b) Al ejecutar `next()` se ejecuta el cuerpo de la función.
  (c) Se ejecuta hasta llegar a la primera declaración `yield` o `return`.
- Aparece la declaración `yield`, la cual entrega el par `{done: ..., value: ...}` y pausa la ejecución hasta la próxima vez que se llama a `next()`.

Estructura y semántica

- Los generadores son **métodos** (¡no objetos!) definidos mediante la declaración `function*`.
- Siguen la estructura de cualquier otra función, pero:
  (a) Al llamarlos, se devuelve un objeto iterable.
  (b) Al ejecutar `next()` se ejecuta el cuerpo de la función.
  (c) Se ejecuta hasta llegar a la primera declaración `yield` o `return`.
- Aparece la declaración `yield`, la cual entrega el par `{done: ..., value: ...}` y pausa la ejecución hasta la próxima vez que se llama a `next()`.

Ejemplos

- `ejemplo1.js`: construcción y llamado de un generador básico. Se comprueba que los generadores devuelven iteradores.
- `ejemplo2.js`: llamando a un generador desde un generador usando `yield*`.

Ejemplos

- `ejemplo1.js`: construcción y llamado de un generador básico. Se comprueba que los generadores devuelven iteradores.
- `ejemplo2.js`: llamando a un generador desde un generador usando `yield*`.

Ejemplos

- `ejemplo1.js`: construcción y llamado de un generador básico. Se comprueba que los generadores devuelven iteradores.
- `ejemplo2.js`: llamando a un generador desde un generador usando `yield*`.

Comentarios

- Son una herramienta muy poderosa pues permiten el trabajo en paralelo, y pausar la ejecución de bloques de código hasta que se cumplan condiciones:

```
function* task(){
  console.log("Ejecucion iniciada");
  yield 1;
  console.log("Ejecucion retomada");
  yield 2;
  console.log("Ejecucion finalizda");
  yield 3;
}

var t = task();
console.log("-> No pasa nada. Hay que ejecutar next() para llamar al cuerpo.");
t.next();
console.log("-> No pasara nada hasta que llamemos de nuevo a next()...");
t.next();
console.log("-> El generador debe esperar...");
t.next();
```

Comentarios

- Usando `yield*` puede llamarse a un generador dentro de un generador.
- `Symbol.iterator` puede definirse mediante un generador.
- El generador dispone del método `throw()` para recibir una excepción, la cual se maneja mediante `try { ... } catch (...) {...}` al interior de su cuerpo.

Comentarios

- Usando `yield*` puede llamarse a un generador dentro de un generador.
- `Symbol.iterator` puede definirse mediante un generador.
- El generador dispone del método `throw()` para recibir una excepción, la cual se maneja mediante `try { ... } catch (...) {...}` al interior de su cuerpo.

Comentarios

- Usando `yield*` puede llamarse a un generador dentro de un generador.
- `Symbol.iterator` puede definirse mediante un generador.
- El generador dispone del método `throw()` para recibir una excepción, la cual se maneja mediante `try { ... } catch (...) {...}` al interior de su cuerpo.

Conclusiones

Conclusiones

Los iteradores y generadores son herramientas fundamentales de ES6 para el trabajo con JavaScript asíncrono.

- Los iteradores entregan una abstracción para poder consumir datos de distintos tipos de colecciones.
- Los generadores permiten:
  - compactar la sintaxis de iteradores.
  - generar funciones pausables.
  - expandir la naturaleza asíncrona de JavaScript.

Conclusiones

Los iteradores y generadores son herramientas fundamentales de ES6 para el trabajo con JavaScript asíncrono.

- Los iteradores entregan una abstracción para poder consumir datos de distintos tipos de colecciones.
- Los generadores permiten:
  - compactar la sintaxis de iteradores.
  - generar funciones pausables.
  - expandir la naturaleza asíncrona de JavaScript.

Conclusiones

Los iteradores y generadores son herramientas fundamentales de ES6 para el trabajo con JavaScript asíncrono.

- Los iteradores entregan una abstracción para poder consumir datos de distintos tipos de colecciones.
- Los generadores permiten:
  - compactar la sintaxis de iteradores.
  - generar funciones pausables.
  - expandir la naturaleza asíncrona de JavaScript.

Referencias y más información

Las fuentes empleadas son una excelente referencia para buscar más información:

Webcasts:

- JavaScript ES6 - Iterators and Generators
- ES2015 Iterators and Generators - Dan Shappir
- Javascript ES6 - Iterators

Artículos de sitios web:

- Iterables and iterators in ECMAScript 6.
- The Basics Of ES6 Generators.

Documentación oficial:

- Iteration protocols - JavaScript | MDN.
- function* - JavaScript | MDN.

Bibliografía escrita:

- B. Syed, Beginning Node.js, 1st edition, Apress, 2014.