Programación Asíncrona

Introducción

- El intérprete de javascript ejecuta el código **en una sola hebra**
- **NO** permite **concurrencia**
- Si la hebra se bloquea, **TODO** el programa se bloquea!

Introducción

- Abre una página cualquiera con el navegador
- Teclea en la consola:

```
for (let i = 1e10; i--;);
```

- E intenta interactuar con la página

Introducción

- ¿Qué pasa si tenemos operaciones largas de entrada/salida?

```
const result = syncHttpGet('/me'); // 300ms
```

```
const button = document.querySelector('#button');

button.addEventListener('click', () => {
  alert('Clicked!');
});

console.log('* ready');
```

Introducción

- Una sola hebra = decisión de diseño
  - Elimina completamente la **concurrencia** de código
  - Facilita **MUCHO** la escritura de programas
  - Mantiene la concurrencia de I/O
  - Transparente para el programador

Callbacks

Callbacks

- Un **callback** es
  - Una **función**
  - Que **definimos nosotros**
  - Pero que **será ejecutada** por **otro agente**
  - Con **parámetros** que describen el suceso al que estaba asociado

```
const callback = () => alert('hi');

setTimeout(callback, 100);
```

Callbacks

- **TODA** la asincronía en JS se basa en callbacks
  - las demás técnicas son patrones sobre callbacks
- Mecanismo de “**bajo nivel**”

Callbacks

- **DOS** limitaciones importantes respecto a las funciones
  - **NO** se puede recuperar su *valor de retorno*
  - **NO** se pueden capturar sus *excepciones*

```
function delaySum(a, b) {
  setTimeout(() => a + b, 100);
}

const result = delaySum(12, 90);
console.log(result); // ???
```

Callbacks

- La **única** manera de “devolver” un valor desde un callback es **llamando a otro callback**
  - La asincronía es *contagiosa*
  - En cuanto un valor es asíncrono, **todo el código que lo utilice** va a ser asíncrono también

```
function delaySum(a, b, callback) {
  setTimeout(() => callback(a + b), 100);
}

delaySum(12, 90, (result) => {
  console.log(result);
});
```

```
function delaySum(a, b, callback) {
  setTimeout(() => callback(a + b), 100);
}

delaySum(12, 90, (result) => {
  console.log(result);
});
```

```
function delaySum(a, b, callback) {
  setTimeout(() => callback(a + b), 100);
}

delaySum(12, 90, (result) => {
  console.log(result);
});
```

```
function delayDivision(a, b, callback) {
  setTimeout(() => {
    if (b === 0) throw new Error('Div by 0!');
    callback(a / b);
  }, 100);
}

try {
  delayDivision(12, 0, (result) => {
    console.log(result);
  });
} catch(e) {
  console.log('Safely captured:', e.message);
}
```

Callbacks

- El **acuerdo general** (sobre todo en node.js) es:
  - Los callbacks **NUNCA** levantan excepción
  - Si hay algún error, se pasa **como primer parámetro** al **callback**
  - Si todo va bien, el primer parámetro se deja a **null**

```
function delayDiv(a, b, callback) {
  setTimeout(() => {
    if (b === 0) {
      callback(new Error('Div by 0!'))
    } else {
      callback(null, a / b);
    }
  }, 100);
}

delayDiv(12, 0, (err, result) => {
  if (err) {
    console.log('Safely captured:', err.message);
  } else {
    console.log(result);
  }
});
```

```
function delayDiv(a, b, callback) {
  setTimeout(() => {
    if (b === 0) {
      callback(new Error('Div by 0!'))
    } else {
      callback(null, a / b);
    }
  }, 100);
}

delayDiv(12, 0, (err, result) => {
  if (err) {
    console.log('Safely captured:', err.message);
  } else {
    console.log(result);
  }
});
```

Callbacks

- Esto es, en esencia, **todo lo que hay que saber** sobre programación asíncrona en javascript

Callbacks

- Supongamos que tenemos cuatro *funciones asíncronas*
  - `getPlayers(callback)`
  - `throwDice(callback)`
  - `savePlayerScore(score, callback)`
  - `getScoreBoard(callback)`

Operaciones Sobre Listas

- `getPlayers(callback)`
  - invoca **callback** con un **array** de nombres de jugadores
    - `callback(err, players)`

Operaciones Sobre Listas

- `throwDice(callback)`
  - invoca **callback** con **un número** aleatorio entre 1 y 6
    - `callback(err, number)`

Operaciones Sobre Listas

- `savePlayerScore(score, callback)`
  - **almacenamos** la puntuación de un jugador
    - `{ player: 'nombre', score: [4, 3] }`
  - invoca **callback** cuando la operación finalice
    - `callback(err)`

Operaciones Sobre Listas

- `getScoreBoard(callback)`
  - invoca **callback** con la **lista de puntuaciones** de todos los jugadores
    - `callback(err, scores)`

Ejercicio: Callbacks

- `ejercicios/e1-callback/index.js`
- Vamos a implementar el flujo completo para **el primer jugador del array**:
  - Primero solicitamos la **lista de jugadores y guardamos el nombre del primer jugador**
  - Después, **tiramos dos dados** (uno tras otro)
  - Cuando tengamos las dos tiradas, **guardamos** la puntuación para el primer jugador
  - Después de guardarla, **solicitamos la lista de puntuación** y la mostramos en la consola

```
getPlayers((err, [player]) => {
  throwDice((err, dice1) => {
    throwDice((err, dice2) => {
      const score = { player, score: [dice1, dice2] };
      savePlayerScore(score, (err) => {
        getScoreBoard(console.log);
      });
    });
  });
});
```

Ejercicio: Callbacks

- Los **callbacks** reciben como primer parámetro:
  - **null** si no hubo ningún error
  - una instancia de **Error**
- Modifica el código para que, si se recibe un error, el error se muestre por consola y se pare la ejecución

```
getPlayers((err, [player]) => {
  if (err) {
    console.log(err);
  } else {
    throwDice((err, dice1) => {
      if (err) {
        console.log(err);
      } else {
        throwDice((err, dice2) => {
          if (err) {
            console.log(err);
          } else {
            const score = { player, score: [dice1, dice2] };
            savePlayerScore(score, (err) => {
              if (err) {
                console.log(err);
              } else {
                getScoreBoard((err, scoreBoard) => console.log(scoreBoard));
              }
            });
          }
        });
      }
    });
  }
});
```

```
getPlayers((err, [player]) => {
  if (err) {
    console.log(err);
  } else {
    throwDice((err, dice1) => {
      if (err) {
        console.log(err);
      } else {
        throwDice((err, dice2) => {
          if (err) {
            console.log(err);
          } else {
            const score = { player, score: [dice1, dice2] };
            savePlayerScore(score, (err) => {
              if (err) {
                console.log(err);
              } else {
                getScoreBoard((err, scoreBoard) => console.log(scoreBoard));
              }
            });
          }
        });
      }
    });
  }
});
```

redradix

Ejercicio: Callbacks

- Ahora aplica el mismo flujo **para todos los jugadores** del array
  - Primero implementa la solución en **paralelo**
  - Después solución **en serie**
- Llama a **getScoreBoard()** cuando **TODOS** hayan terminado

Observables

Observables

- Tradicionalmente:
  - desacoplar objetos dependientes
- Javascript
  - suscribir varios callbacks a un mismo agente
  - modelar procesos más sofisticado

Observables

- Un **productor**
- Múltiples **consumidores**
- Se comunican mediante **eventos**

```
class Metronome extends Observable {
  constructor(tempo) {
    super();
    this.intervalId = setInterval(() => this.tick(), tempo);
    this.counter = 0;
  }
  tick() {
    this.emit('tick', this.counter++);
  }
}

const m = new Metronome(1000);
m.on('tick', t => console.log('* tick number', t));
```

```
class Metronome extends Observable {
  constructor(tempo) {
    super();
    this.intervalId = setInterval(() => this.tick(), tempo);
    this.counter = 0;
  }
  tick() {
    this.emit('tick', this.counter++);
  }
}

const m = new Metronome(1000);
m.on('tick', t => console.log('* tick number', t));
```

```
class Metronome extends Observable {
  constructor(tempo) {
    super();
    this.intervalId = setInterval(() => this.tick(), tempo);
    this.counter = 0;
  }
  tick() {
    this.emit('tick', this.counter++);
  }
}

const m = new Metronome(1000);
m.on('tick', t => console.log('* tick number', t));
```

```
class Metronome extends Observable {
  constructor(tempo) {
    super();
    this.intervalId = setInterval(() => this.tick(), tempo);
    this.counter = 0;
  }
  tick() {
    this.emit('tick', this.counter++);
  }
}

const m = new Metronome(1000);
m.on('tick', t => console.log('* tick number', t));
```

Operaciones Sobre Listas

- `on(event, callback)`
  - **event**: string que identifica al evento
  - **callback**: la función que se asocia con ese evento

Operaciones Sobre Listas

- `off(event, callback)`
  - **elimina** la asociación entre **callback** y **event**
  - Es decir: **callback** ya no se ejecutará cuando se emita **event**

Operaciones Sobre Listas

- `emit(event, ...args)`
  - **emite** el evento **event** con los parámetros **args**
  - todos los **callbacks** asociados a **event** serán ejecutados con los parámetros **args**

Ejercicio: Observable

- Implementa la clase **`Observable`**
  - Con sus tres métodos **on**, **off** y **emit**
  - Para que funcione el ejemplo anterior

```
class Observable {
  constructor() {

  }
  on(event, cb) {

  }
  off(event, cb) {

  }
  emit(event, ...payload) {

  }
}
```

Observables

- Probablemente *el patrón más importante en JS*
  - En *node.js* se llama **EventEmitter**
  - Fundamento para otras estructuras más complejas
    - Queues
    - Streams
    - ...

Ejercicio: Observable

- Crea una clase **Dados** que emite **valores entre 1 y 6** emitiendo el evento **tirada** en un **intervalo aleatorio** entre 100 y 500 ms
- Crea dos instancias de **Dados**
- Escribe el código necesario para loguear **pares** de tiradas

Promesas

Promesas

- Una abstracción de nivel mucho más alto
- **MUY** componible
- **MUY** popular

Promesas

- Una promesa representa **un valor futuro**

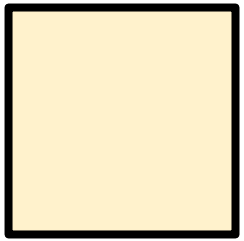
Promesas

- Una promesa es un **objeto** que hace de **intermediario**
  - entre el **productor** de un valor
  - y sus **consumidores**
- Para simplificar la gestión de procesos asíncronos

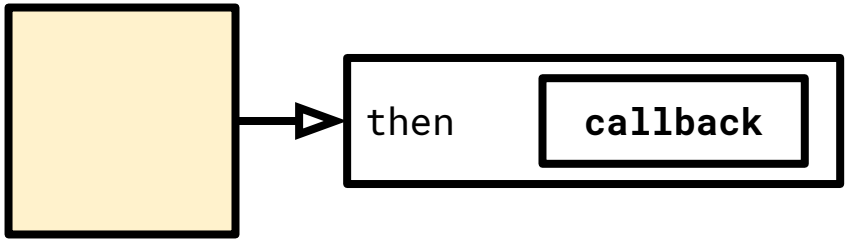
Promesas

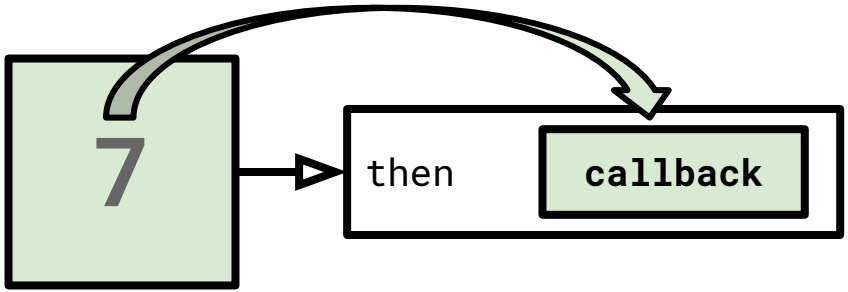
- Una promesa es **una caja** que **encierra** un **valor**

7



redradix





Promesas

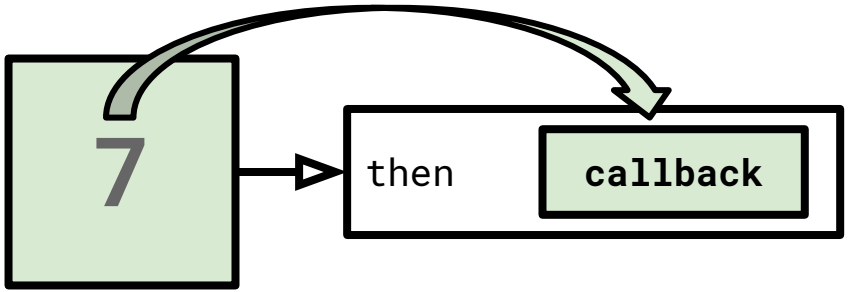
- Hay **tres** maneras de crear una promesa
  - `Promise.resolve(value)`
  - `Promise.reject(error)`
  - `new Promise(...)`

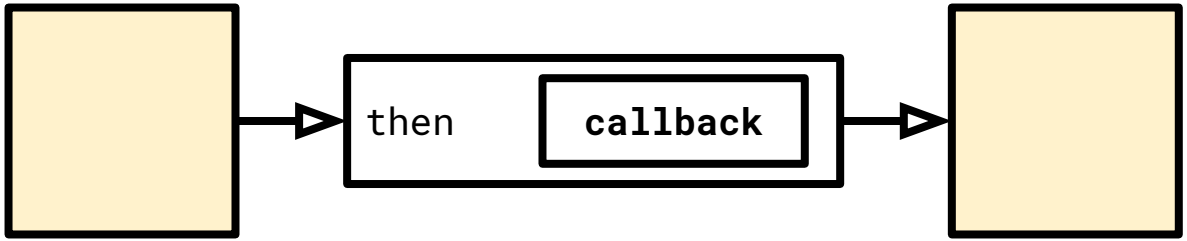
Promesas

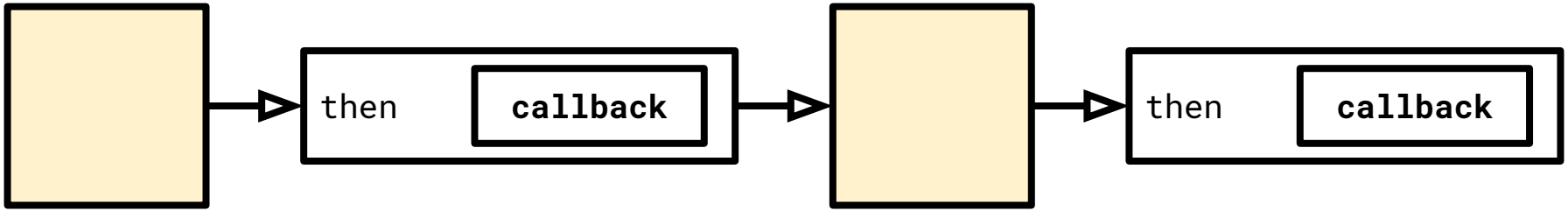
- `Promise.resolve(value)`
  - Crea una promesa **resuelta**
  - Los **callbacks** de **.then** se ejecutan **inmediatamente**

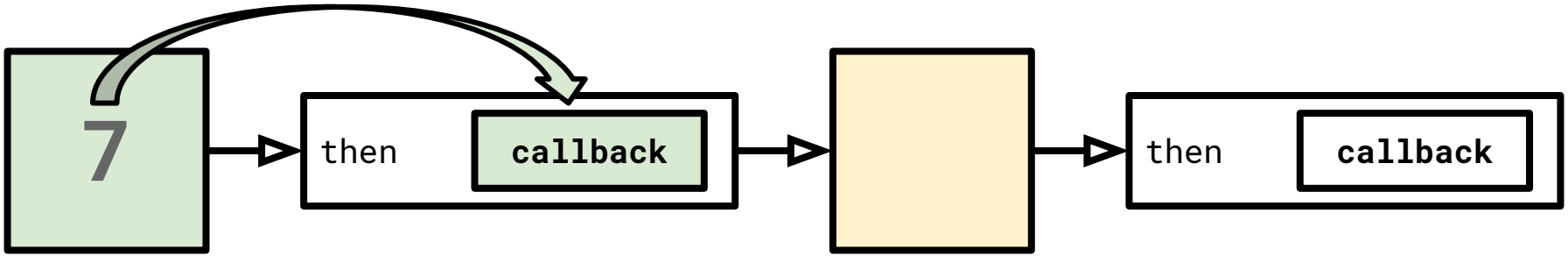
Promesas

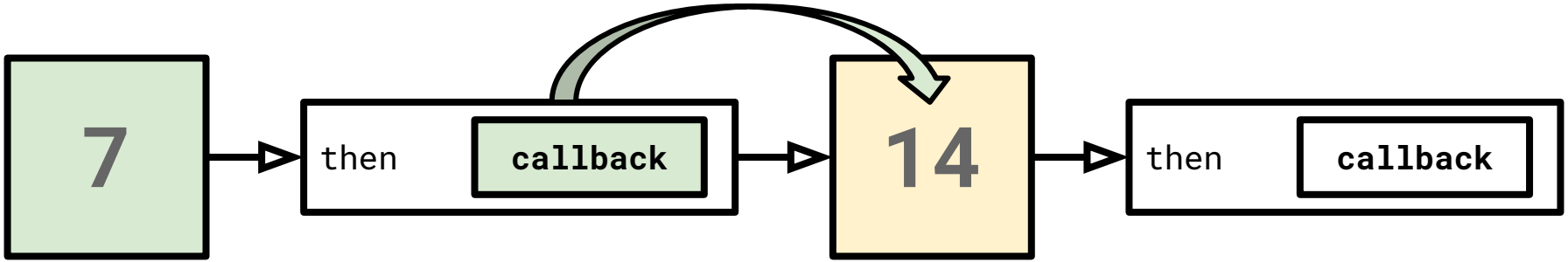
```
// Creamos caja con un 7
let promise = Promise.resolve(7)



// Abrimos caja
promise.then(result => {
    console.log(result)       //7
})
```

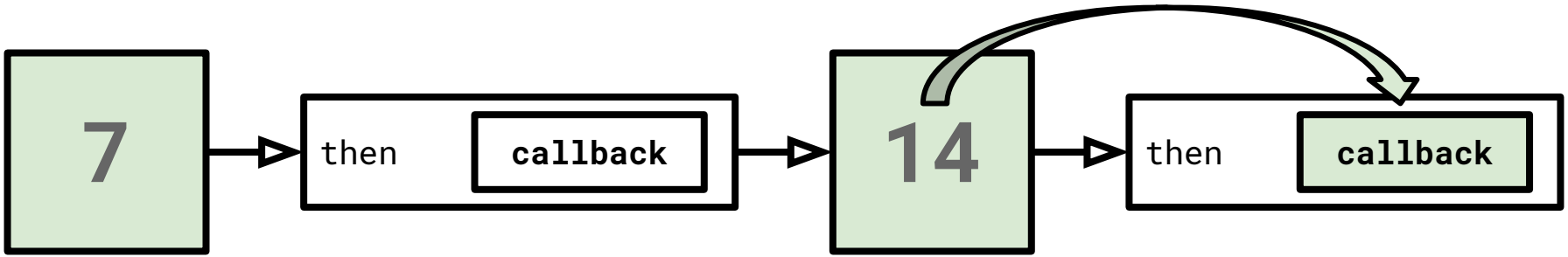












Promesas

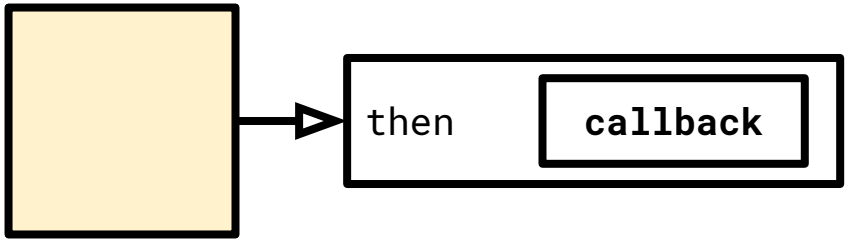
```
let promise = Promise.resolve(7)
```

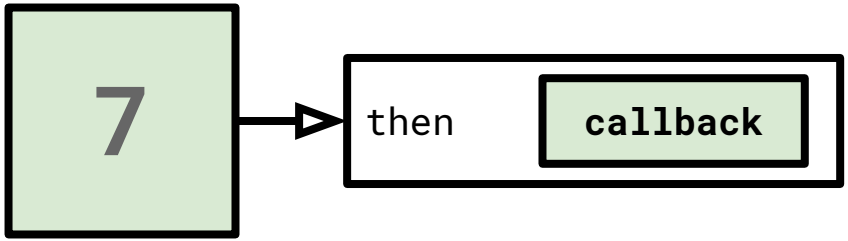
```
let promise2 = promise.then( result => {
    return result * 2
})
```

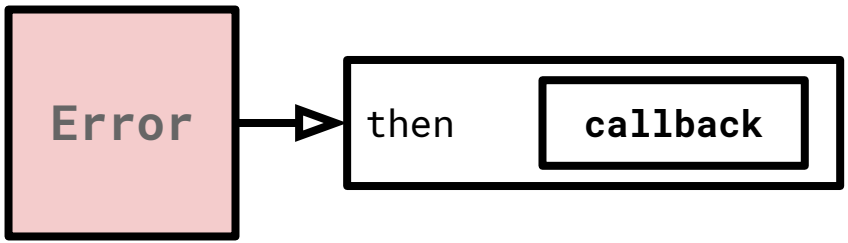
```
promise2.then(result => {
    console.log(result)        //14
})
```

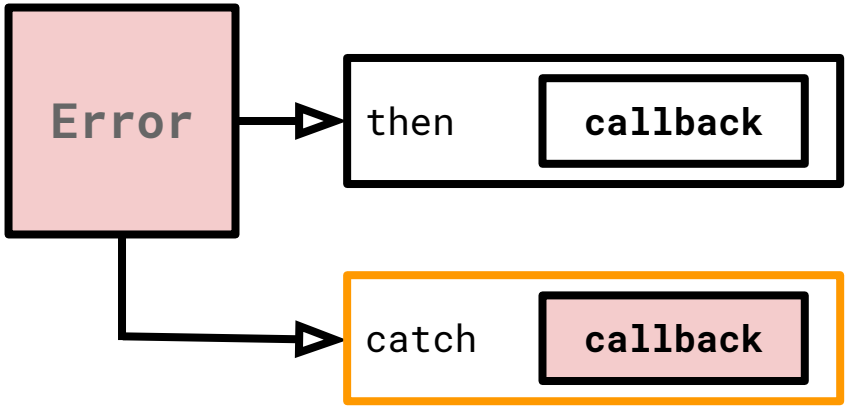
Promesas

- Una promesa tiene 3 estados posibles
  - `pending`
  - `fulfilled`
  - `rejected`



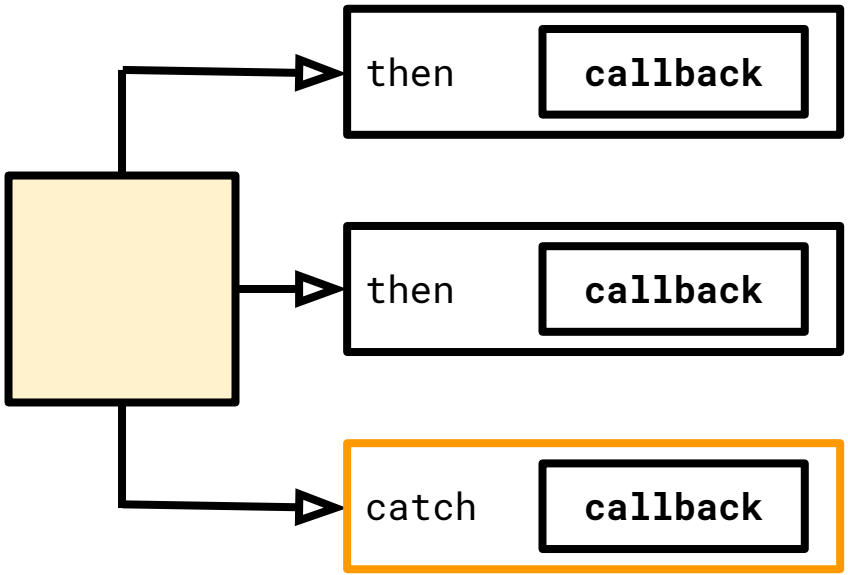


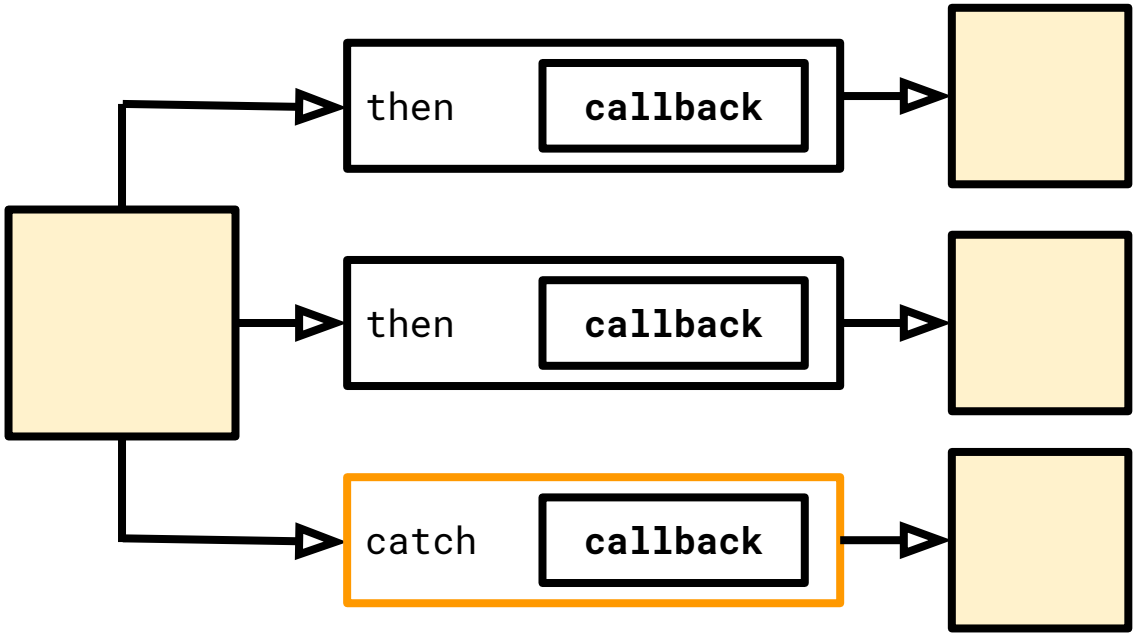


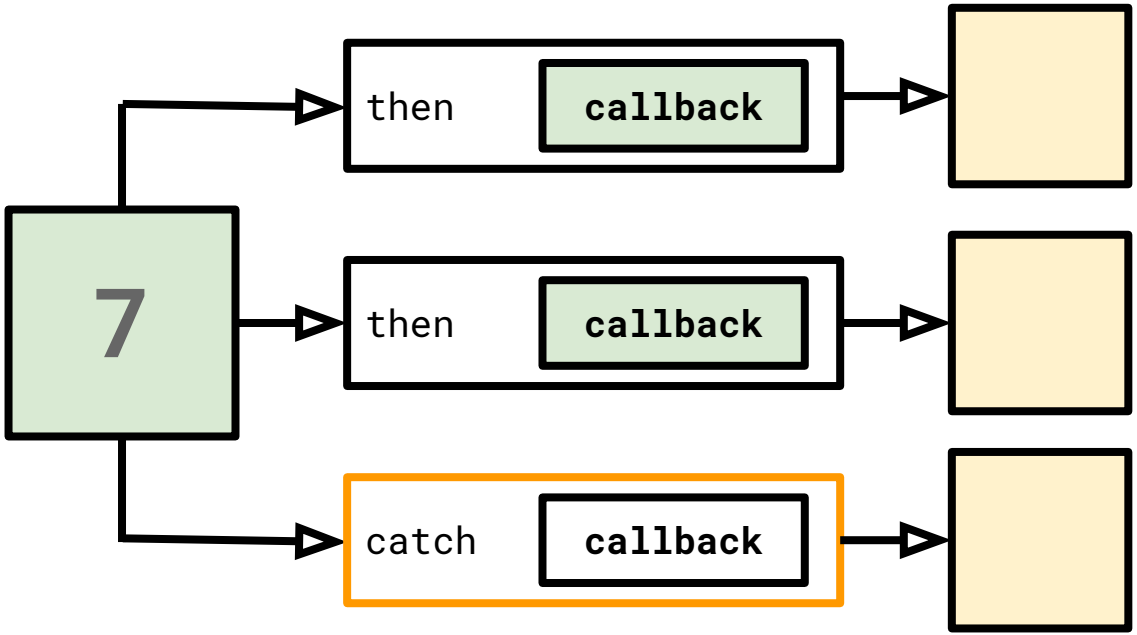


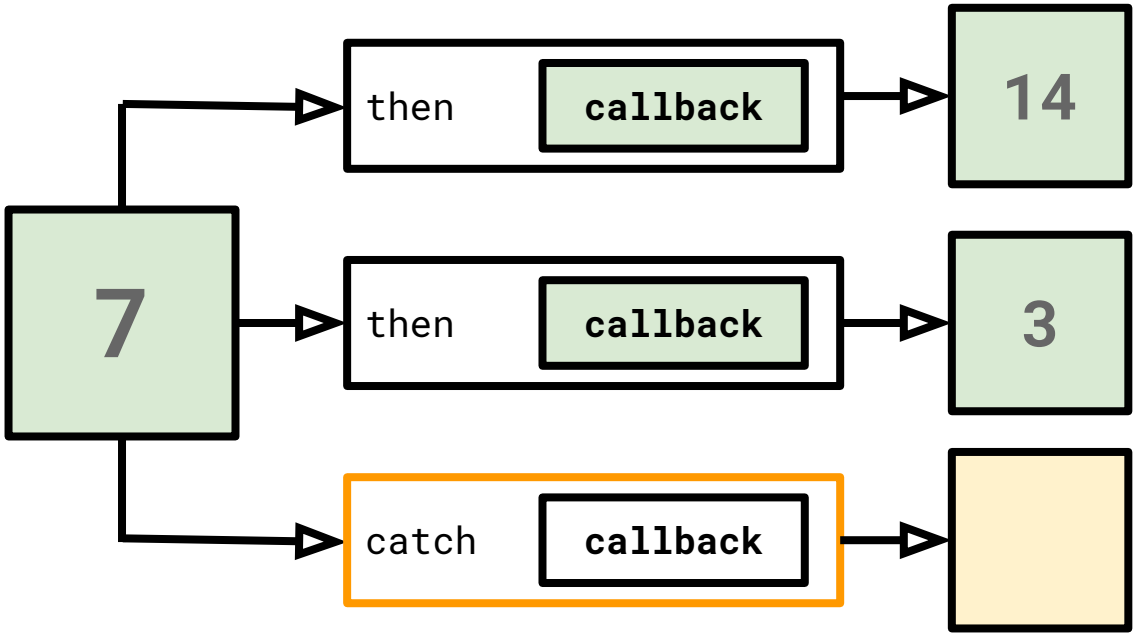
Promesas

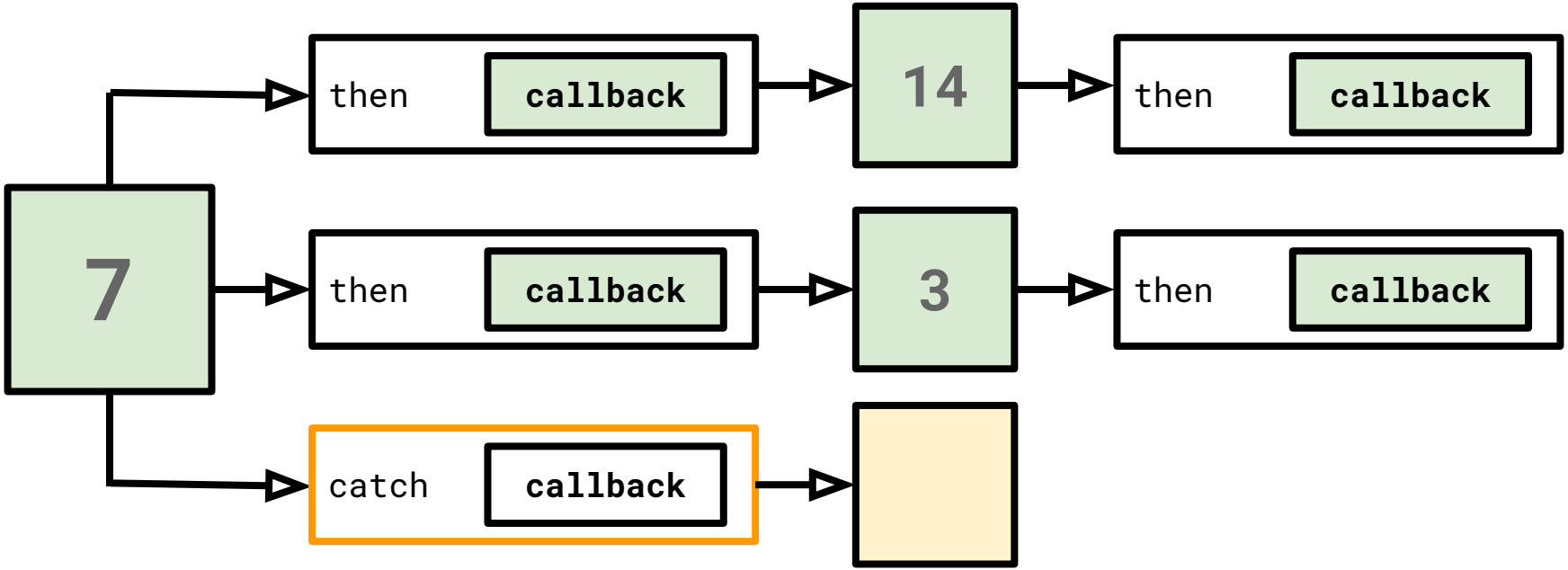
- Una promesa tiene tres estados:
  - pendiente
  - resuelta
  - rechazada
- Cuando una promesa se **resuelve** o se **rechaza**, **no puede volver a cambiar de estado**
  - se queda resuelta o rechazada para siempre





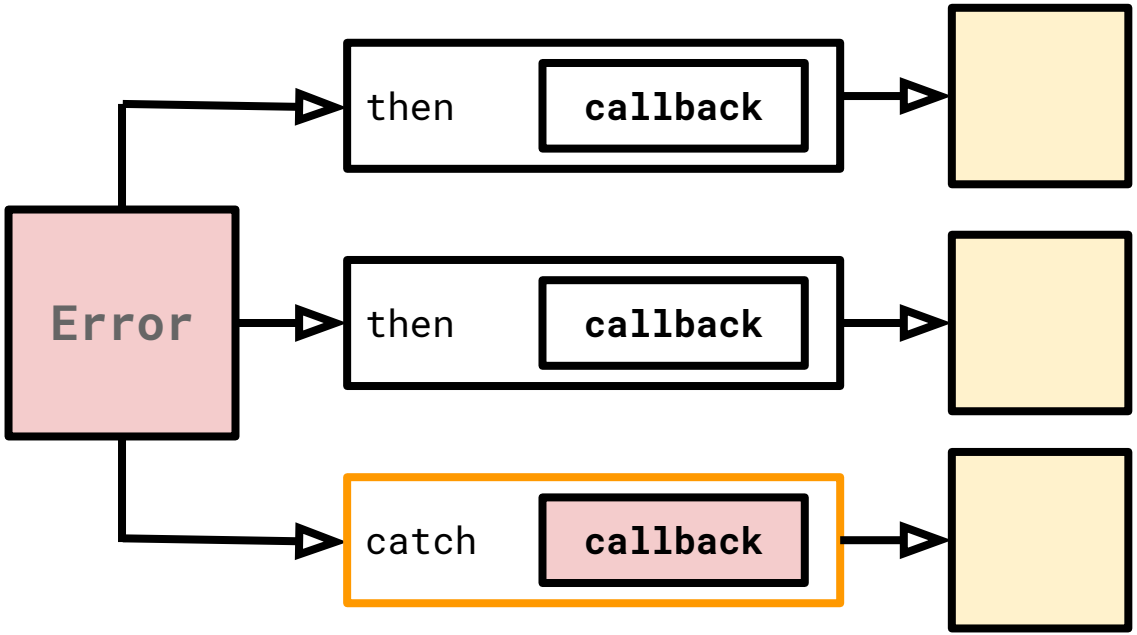


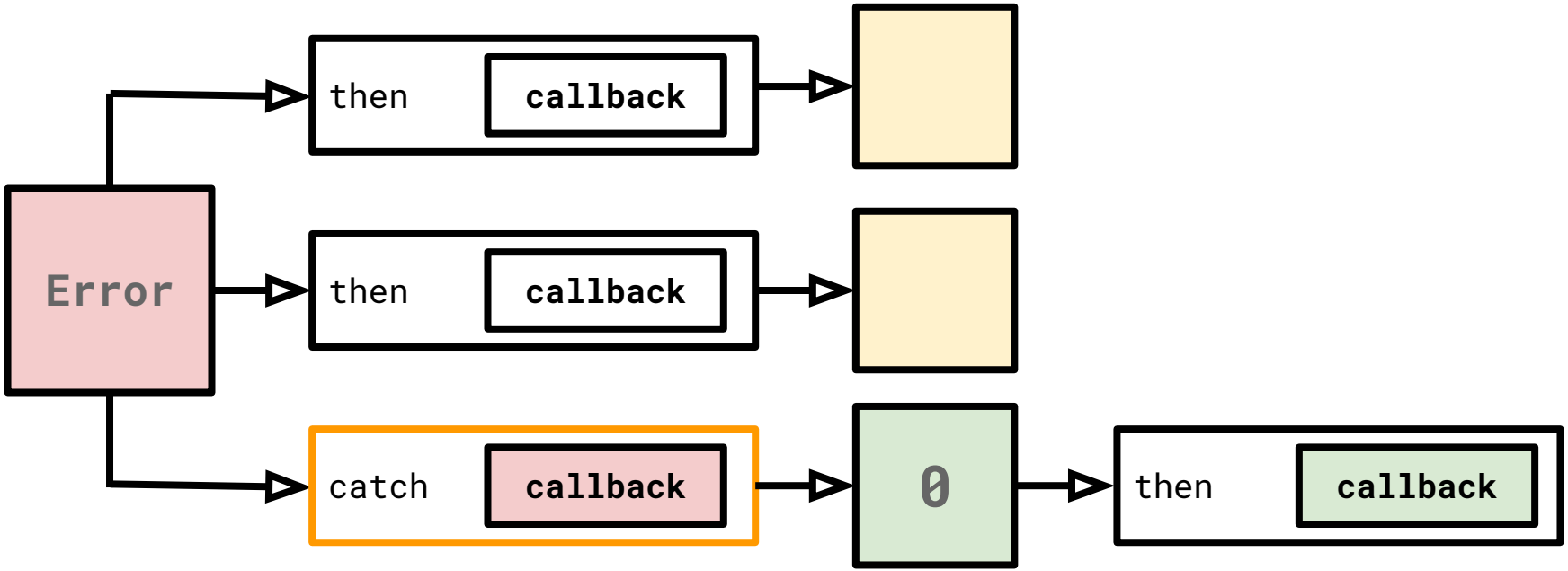

7
then
callback
14
then
callback
3
catch
callback


7
then
callback
14
then
callback
then
callback
3
then
callback
catch
callback

Promesas

- Las llamadas a **.then()** y a **.catch()** devuelven **una nueva promesa**
- Que representa el **valor de retorno** de sus **callbacks**
- El callback the **.then()** se ejecuta cuando la promesa se **resuelve**
- El callback the **.catch()** se ejecuta cuando la promesa se **rechaza**




then
callback
Error
then
callback
catch
callback
0
then
callback

Promesas

- Hay **tres** maneras de crear una promesa
  - `Promise.resolve(value)`
  - `Promise.reject(error)`
  - `new Promise(...)`

Promesas

- `Promise.resolve(value)`
  - Crea una promesa **resuelta**
  - Los **callbacks** de **.then** se ejecutan **inmediatamente**

```
const p = Promise.resolve('ready');
p.then(console.log);
```

```
const p = Promise.resolve('ready');
p.catch(console.log); //?
```

Promesas

- `Promise.reject(error)`
  - Crea una promesa **rechazada**
  - Los **callbacks** de **.catch** se ejecutan **inmediatamente**

```
const p = Promise.reject(new Error('doomed from the start'));
p.catch(console.log);
```

```
const p = Promise.reject(new Error('doomed from the start'));
p.then(console.log);
```

Promesas

- `new Promise(callback)`
  - Crea una promesa **pendiente**
  - El callback se ejecuta **con delay 0**
  - **callback** recibe dos parámetros
    - **resolve**: callback de resolución
    - **reject**: callback de rechazo

```
const p = new Promise((resolve, reject) => {
  setTimeout(() => resolve('resolved'), 1000)
});

p.then(console.log);

console.log('antes o después?')
```

```
const p = new Promise((resolve, reject) => {
  setTimeout(() => resolve(new Error('rejected'), 1000)
});

p.catch(console.log);

console.log('antes o después?')
```

Ejercicio Promesas

- Escribe una función **throwOneCoin** que devuelva una promesa que represente el lanzamiento de una moneda.
  - La moneda tarda 2 segundos en caer
  - **50%** de las veces, la promesa se **resuelve** y se muestra **“cruz!”** por la consola
  - **50%** de las veces, la promesa se **rechaza** y se muestra **“cara...”** por la consola

```
function getDate(cb) {
  setTimeout(() => cb(Date.now()), 100);
}

getDate((date) => {
  getDate((date2) => {
    getDate((date3) => {
      // seguimos por aquí
    });
    // ???
  });
});
```

```
function getDate() {
  return new Promise((resolve) => {
    setTimeout(() => resolve(Date.now()), 100)
  });
}
```

```
function getDate() {
  return new Promise((resolve) => {
    setTimeout(() => resolve(Date.now()), 100)
  });
}

const datePromise = getDate();
// seguimos por aquí!
```

```
function getDate() {
  return new Promise((resolve) => {
    setTimeout(() => resolve(Date.now()), 100)
  });
}

const datePromise = getDate();
const datePromise2 = getDate();
const datePromise3 = getDate();
// seguimos por aquí!
```

```
function getDate() {
  return new Promise((resolve) => {
    setTimeout(() => resolve(Date.now()), 100)
  });
}

getDate()
    .then(() => getDate())
    .then(() => getDate())
```

```
function futureValue(n) {
  return new Promise(
    resolve => setTimeout(() => resolve(n), 1000)
  );
}

futureValue(1)
  .then(v => futureValue(v + 1))
  .then(v => futureValue(v + 1))
  .then(console.log); // ???
```

Promesas

- `.then(...)`
  - Crear *secuencias de operaciones asíncronas*
  - Manteniendo **un flujo de ejecución claro**
  - Sin necesidad de indentar cada paso

Promesas

- Una promesa se considera **rechazada** si **se levanta una excepción** o si se llama al callback **reject()**

```
const p1 = new Promise((resolve, reject) => {
  throw new Error('Oh, noes!');
});

p1.catch(e => console.log('Captured:', e.message));
```

Promesas

- `.catch(rejectCallback)`
  - Devuelve **una promesa**
  - La promesa devuelta se comporta igual que la devuelta por **.then()**
  - El valor de **resolución** será el valor retornado por **rejectCallback**

```
const p1 = new Promise((resolve, reject) => {
  throw new Error('Oh, noes!');
});

p1.catch((e) => {
  console.log('Captured:', e.message);
  return e;
})
  .then(
    () => console.log('All good!') //????
  );
```

```
const p1 = new Promise((resolve, reject) => {
  throw new Error('Oh, noes!');
});

p1
  .then(() => console.log('1...'))
  .then(() => console.log('2...'))
  .then(() => console.log('3...'))
  .catch(() => console.log('Something bad happened'));
///?
```

```
const p1 = new Promise((resolve, reject) => {
  throw new Error('Oh, noes!');
});

p1
  .then(() => console.log('1...')) // no rejectCallback -> rechazada!
  .then(() => console.log('2...')) // no rejectCallback -> rechazada!
  .then(() => console.log('3...')) // no rejectCallback -> rechazada!
  .catch(() => console.log('Something bad happened'));
```

```
const p1 = new Promise((resolve, reject) => {
  throw new Error('Oh, noes!');
});

p1
  .then(() => console.log('1...'))
  .then(() => console.log('2...'))
  .then(() => console.log('3...'))
  .catch(() => console.log('Something bad happened'))
  .then(() => console.log('Everything under control'));
// ???
```

Promesas

- En una cadena de promesas
  - Los errores se **propagan hacia abajo**
  - Si se captura el error, la cadena **se resuelve con normalidad** a partir de ese punto
- Facilita el manejo de errores en procesos asíncronos

Ejercicio: SWAPI

- Busca las 5 primeras personas en la API https://swapi.co/
-
- Imprime un array con los nombres de esas personas y calcula la media de estatura

Promesas

- `Promise.all([prom1, prom2, prom3, …])`
  - Devuelve **una nueva promesa**
  - Se resuelve cuando se **hayan resuelto todas** las promesas
  - **Valor de resolución**: valores de resolución de cada promesa
  - Si **una promesa es rechazada**, la promesa devuelta **se rechaza con el mismo error**

```
function futureValue(n, ms) {
  return new Promise(resolve => setTimeout(() => resolve(n), ms));
}

const p = Promise.all([
  futureValue(1, 100),
  futureValue(2, 200),
  futureValue(3, 300),
  futureValue(4, 400),
]);

p.then(console.log); // [ 1, 2, 3, 4 ]
```

```
function futureValue(n, ms) {
  return new Promise(resolve => setTimeout(() => resolve(n), ms));
}

function futureFail(msg, ms) {
  return new Promise(
    (resolve, reject) => setTimeout(() => reject(msg), ms)
  );
}

const p = Promise.all([
  futureValue(1, 100),
  futureValue(2, 200),
  futureFail('Bad luck', 100),
]);

p.catch(console.log); // Bad luck
```

Ejercicio: SWAPI

- Resuelve el ejercicio anterior utilizando Promise.all

Promesas

- `Promise.race([prom1, prom2, prom3, …])`
  - Devuelve **una nueva promesa**
  - **Refleja** el valor de la **primera promesa** que se **resuelva** o **rechace**

```
function futureValue(n, ms) {
  return new Promise(resolve => setTimeout(() => resolve(n), ms));
}

function futureFail(msg, ms) {
  return new Promise(
    (resolve, reject) => setTimeout(() => reject(msg), ms)
  );
}

const p = Promise.race([
  futureValue(1, 100),
  futureValue(2, 200),
  futureFail('Bad luck', 200),
]);

p.then(console.log, console.log); // 1
```

```
function futureValue(n, ms) {
  return new Promise(resolve => setTimeout(() => resolve(n), ms));
}

function futureFail(msg, ms) {
  return new Promise(
    (resolve, reject) => setTimeout(() => reject(msg), ms)
  );
}

const p = Promise.race([
  futureValue(1, 100),
  futureValue(2, 200),
  futureFail('Bad luck', 50),
]);

p.then(console.log, console.log); // Bad luck
```

Ejercicio: Promesas

- Implementa *mapPromise(fn, promisesOrValues)*
  - Aplica **fn** al *valor* de cada promesa de la lista
  - En paralelo
  - Devuelve una **promesa**
  - Que se resuelve a una lista de **valores**

Ejercicio: Promesas

- Implementa *mapSeriesPromise(fn, promisesOrValues)*
  - Sejemante a *mapPromise*
  - Pero **la iteración sucede en serie**

Corrutinas

Corrutinas

- Los *generadores* tienen una cualidad única:
  - **detener el flujo de ejecución** en cualquier momento
  - y luego **continuar donde fueron interrumpidos**
  - sin necesidad de utilizar callbacks!

```
function* counter() {
  console.log('block 1');
  yield 1;
  console.log('block 2');
  yield 2;
  console.log('block 3');
  yield 3;
}

const c = counter();
console.log(c.next().value);
console.log(c.next().value);

setTimeout(() => console.log(c.next().value), 1000);
```

Corrutinas

- ¿Cómo podríamos sacar provecho?
  - detener la ejecución cuando llamamos a un método asíncrono...
  - ... y retomarla cuando el método haya terminado

Corrutinas

1. Escribimos nuestro código en un generador
2. Hacemos **yield** de **promesas**
3. Cuando la promesa se **resuelve**, se retoma la ejecución
4. Si la promesa se **rechaza**, levantamos **excepción**

```
function futureValue(n, ms) {
  return new Promise(resolve => setTimeout(() => resolve(n), ms));
}

function* () {
  console.log('vamos a parar para esperar al valor');
  const value = yield futureValue(10, 1000);
  console.log('el valor es:', value);
  const double = yield futureValue(value * 2, 1000);
  console.log('el doble del valor es:', double);
}
```

Corrutinas

Necesitamos una función que:

1. Reciba el generador con nuestro código
2. Se encargue de instanciarlo
3. Llame a **.next()**
4. Recupere la **promesa yieldeada** y esperar
    a. Si se **rechaza**, levanta una excepción
    b. Si se **resuelve**, pasa el valor de vuelta al generador
5. GOTO 3

```
function futureValue(n, ms) {
  return new Promise(resolve => setTimeout(() => resolve(n), ms));
}

const fn = co(function* () {
  console.log('vamos a parar para esperar al valor');
  const value = yield futureValue(10, 1000);
  console.log('el valor es:', value);
  const double = yield futureValue(value * 2, 1000);
  console.log('el doble del valor es:', double);
});

fn();
```

Ejercicio: Corrutinas

- Implementa la función co(generator)
  - Recibe un generador
  - Devuelve una función
  - Al ejecutarla, avanza el generador paso a paso, esperando a las promesas yieldeadas

Corrutinas

- Para gestionar errores podemos utilizar **try/catch**

```
co(function* () {
  const value = yield futureValue(10, 1000);
  try {
    yield futureFail('boom!', 100);
  } catch (err) {
    console.log('Captured:', err);
  }
})();
```

Corrutinas

- Podemos utilizar **yield** en cualquier expresión

```
co(function* () {
  const values = {
    one: yield futureValue(1, 100),
    two: yield futureValue(2, 100),
    three: yield futureValue(3, 100)
  };
  console.log(values); // { one: 1, two: 2, three: 3 }
  console.log(
    [yield futureValue('a', 100), yield futureValue('b', 200)]
  ); // ['a', 'b']
})();
```

Corrutinas

- Podemos utilizar los métodos de combinación de promesas

```
co(function* () {
  const values = yield Promise.all([
    futureValue(1, 100),
    futureValue(2, 100),
    futureValue(3, 100)
  ]);
  console.log(values);
  // [1, 2, 3] after 100ms
})();
```

Corrutinas

- El **valor de retorno** de la corrutina es **una promesa**
  - Si el generador levanta una excepción no manejada, la promesa se **rechazará**
  - Si todo va bien, la promesa se **resolverá** con el valor de retorno de la corrutina

```
const asyncFunction = co(function* (param) {
  console.log(param);
  const values = yield Promise.all([
    futureValue(1, 100),
    futureValue(2, 100),
    futureValue(3, 100)
  ]);
  return values;
});

const p = asyncFunction('Hi there');
p.then((values) => {
  console.log(values);  // [1, 2, 3] after 100ms
});
```

Corrutinas

- ES2017 introduce **corrutinas nativas**
  - **async** para declarar la corrutina
  - **await** para esperar a la resolución de promesas

```
(async () => {
  const values = {
    one: await futureValue(1, 100),
    two: await futureValue(2, 100),
    three: await futureValue(3, 100)
  };
  console.log(values); // { one: 1, two: 2, three: 3 }
  console.log([
    await futureValue('a', 100),
    await futureValue('b', 200)
  ]); // ['a', 'b']
})();
```

```
async function processItemCo(item) {
  if (item instanceof Node) {
    const children = await new Promise(res => item.getChildren(res));
    return await mapPromise(processItemCo, children);
  } else if (item.isFinalLeaf()){
    return item.getValue();
  } else {
    return processItemCo(
      await new Promise(res => item.getNextLeaf(res))
    );
  }
}

(async () => {
  const nodes = await new Promise(getRootNodes);
  const leafs = await mapPromise(processItemCo, nodes);
  console.log('leafs', leafs);
})();
```