Asynchronous Programming

Introduction

- Javascript runs the code **in a single thread**
- **DOES NOT** allow real **concurrency!**
- If the thread blocks, **EVERYTHING** blocks!

Introduction

- Open any page with your browser
- Type this in the console:

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

- What happens with everything on the page?

Introduction

- But what happens if we have to perform a long I/O operation?

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

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

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

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

Introduction

- Configure some **condition**
- Which we associate with **a function**
- That **it is NOT executed by us**
- **Delegate its execution** to the platform
- And **we change the natural flow of the program**
  - **out of order** execution

Introduction

- The *interpreter* runs **the code** in a single thread...
- ...but **each I/O process** has **its own thread!**
- ...and interact with the main thread using **callbacks**

Introduction

- Single thread = design decision
  - Removes **parallelism** from the equation
  - **Easiest** model of concurrency: not concurrent at all!
  - Only I/O is concurrent
  - Transparent for the programmer

Callbacks

Callbacks

- A **callback** is
  - A **function**
  - Written by **us**
  - But run by **the platform**
  - Usually with some **parameters** containing relevant info

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

setTimeout(callback, 100);
```

Callbacks

- **ALL** async programming in JS is based on callbacks
  - everything else are just patterns to use callbacks
- **“low level”** mechanism
- Continuation Passing Style (CPS)

Callbacks

- **TWO** very important limitations with async functions:
  - **CAN'T** receive their *return value*
  - **CAN'T** capture their *thrown exceptions*

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

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

Callbacks

- The **only** way of “returning” a value from a callback is by **invoking another callback**
  - Async programming is *contagious*
  - When a value **becomes asynchronous**, all the code that uses it becomes asynchronous too!

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

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

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

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

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

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

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

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

Callbacks

- The **general consensus** (especially in node.js) is:
  - Async code **NEVER** raises exceptions
  - If there is any problem, it **gets passed as the first parameter of the callback**
  - If everything goes well, the first parameter of the callback is set to **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

- This is **everything there is to know about async programming in javascript**

Callbacks

- Suppose we have the following *async functions*:
  - `getPlayers(callback)`
  - `throwDice(callback)`
  - `savePlayerScore(score, callback)`
  - `getScoreBoard(callback)`

Callbacks

- `getPlayers(callback)`
  - invokes **callback** with an **array of player names**
    - `callback(err, players)`

Callbacks

- `throwDice(callback)`
  - invokes **callback** with a random number between 1 and 6
    - `callback(err, number)`

Callbacks

- **`savePlayerScore(score, callback)`**
  - **stores** the **score** of a player, represented as:
    - `{ player: 'name', score: [4, 3] }`
  - invokes **callback** when the operation has finished
    - `callback(err)`

Callbacks

- `getScoreBoard(callback)`
  - invokes **callback** with the list of all the saved scores for all the players
    - `callback(err, scores)`

Exercise: Callbacks

- `exercises/e1-callback/index.js`
- Implement the full cycle for **the first player of the array**:
  - First, ask for the **list of players**
  - Then, throw the dice **twice** (one after the other)
  - Then **save both throws**
  - Finally, **ask for the scoreboard** and log it to the console

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

Exercise: Callbacks

- **callbacks** receives any potential error as the first parameter:
  - **null** if everything went well
  - an instance of **Error** otherwise
- Modify your code to detect any errors, log them to the console and stop the execution

```
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));
              }
            });
          }
        });
      }
    });
  }
});
```

Exercise: Callbacks

- Modify the code to **retry** any operations that **had errors**
  - **extra challenge**: retry a limited number of times (receive the number of times as a parameter)

```
function retry(operation, times) {
  return function aux(...args) {
    const callback = args.pop();
    operation(...args, (err, ...results) => {
      if (err) {
        if (times-- === 0) return callback(err, ...results);
        console.log('* retrying:', err.message);
        aux(...args, callback);
      } else {
        callback(null, ...results);
      }
    });
  };
}
```

```
function retry(operation, times) {
  return function aux(...args) {
    const callback = args.pop();
    operation(...args, (err, ...results) => {
      if (err) {
        if (times-- === 0) return callback(err, ...results);
        console.log('* retrying:', err.message);
        aux(...args, callback);
      } else {
        callback(null, ...results);
      }
    });
  };
}
```

```
function retry(operation, times) {
  return function aux(...args) {
    const callback = args.pop();
    operation(...args, (err, ...results) => {
      if (err) {
        if (times-- === 0) return callback(err, ...results);
        console.log('* retrying:', err.message);
        aux(...args, callback);
      } else {
        callback(null, ...results);
      }
    });
  };
}
```

```
function retry(operation, times) {
  return function aux(...args) {
    const callback = args.pop();
    operation(...args, (err, ...results) => {
      if (err) {
        if (times-- === 0) return callback(err, ...results);
        console.log('* retrying:', err.message);
        aux(...args, callback);
      } else {
        callback(null, ...results);
      }
    });
  };
}
```

```
function retry(operation, times) {
  return function aux(...args) {
    const callback = args.pop();
    operation(...args, (err, ...results) => {
      if (err) {
        if (times-- === 0) return callback(err, ...results);
        console.log('* retrying:', err.message);
        aux(...args, callback);
      } else {
        callback(null, ...results);
      }
    });
  };
}
```

```
function retry(operation, times) {
  return function aux(...args) {
    const callback = args.pop();
    operation(...args, (err, ...results) => {
      if (err) {
        if (times-- === 0) return callback(err, ...results);
        console.log('* retrying:', err.message);
        aux(...args, callback);
      } else {
        callback(null, ...results);
      }
    });
  };
}
```

```
function retry(operation, times) {
  return function aux(...args) {
    const callback = args.pop();
    operation(...args, (err, ...results) => {
      if (err) {
        if (times-- === 0) return callback(err, ...results);
        console.log('* retrying:', err.message);
        aux(...args, callback);
      } else {
        callback(null, ...results);
      }
    });
  };
}
```

```
const retryGetPlayers = retry(getPlayers, 10);
const retryThrowDice = retry(throwDice, 10);
const retrySavePlayerScore = retry(savePlayerScore, 10);
const retryGetScoreBoard = retry(getScoreBoard, 10);
```

```
retryGetPlayers((err, [player]) => {
  retryThrowDice((err, dice1) => {
    retryThrowDice((err, dice2) => {
      const score = { player, score: [dice1, dice2] };
      retrySavePlayerScore(score, (err) => {
        retryGetScoreBoard(console.log);
      });
    });
  });
});
```

```
retry(getPlayers, 2)((err, [player]) => {
  retry(throwDice, 2)(((err, dice1) => {
    retry(throwDice, 2)((err, dice2) => {
      const score = { player, score: [dice1, dice2] };
      retry(savePlayerScore, 1)(score, (err) => {
        retry(getScoreBoard, 1)(console.log);
      });
    });
  }))
});
```

Ejercicio: Callbacks

- Now apply the same flow to **every player of the array**
  - First implement a **parallel** solution
  - Then, implement a **serial** solution
- Call **getScoreBoard()** when **EVERY PLAYER** flow has finished

```
function asyncMap(fn, list, callback) {
  const results = [];
  let remaining = list.length;
  let finished = false;
  list.map((item, i) => {
    fn(item, (err, result) => {
      if (finished) return;
      if (err) {
        finished = true;
        return callback(err, []);
      }
      results.push(result);
      if (--remaining === 0) callback(null, results);
    })
  });
}
```

```
function asyncMap(fn, list, callback) {
  const results = [];
  let remaining = list.length;
  let finished = false;
  list.map((item, i) => {
    fn(item, (err, result) => {
      if (finished) return;
      if (err) {
        finished = true;
        return callback(err, []);
      }
      results.push(result);
      if (--remaining === 0) callback(null, results);
    })
  });
}
```

```
function asyncMap(fn, list, callback) {
  const results = [];
  let remaining = list.length;
  let finished = false;
  list.map((item, i) => {
    fn(item, (err, result) => {
      if (finished) return;
      if (err) {
        finished = true;
        return callback(err, []);
      }
      results.push(result);
      if (--remaining === 0) callback(null, results);
    })
  });
}
```

```
function asyncMap(fn, list, callback) {
  const results = [];
  let remaining = list.length;
  let finished = false;
  list.map((item, i) => {
    fn(item, (err, result) => {
      if (finished) return;
      if (err) {
        finished = true;
        return callback(err, []);
      }
      results.push(result);
      if (--remaining === 0) callback(null, results);
    })
  });
}
```

```
function asyncMap(fn, list, callback) {
  const results = [];
  let remaining = list.length;
  let finished = false;
  list.map((item, i) => {
    fn(item, (err, result) => {
      if (finished) return;
      if (err) {
        finished = true;
        return callback(err, []);
      }
      results.push(result);
      if (--remaining === 0) callback(null, results);
    })
  });
}
```

```
function asyncMap(fn, list, callback) {
  const results = [];
  let remaining = list.length;
  let finished = false;
  list.map((item, i) => {
    fn(item, (err, result) => {
      if (finished) return;
      if (err) {
        finished = true;
        return callback(err, []);
      }
      results.push(result);
      if (--remaining === 0) callback(null, results);
    })
  });
}
```

```
function asyncMap(fn, list, callback) {
  const results = [];
  let remaining = list.length;
  let finished = false;
  list.map((item, i) => {
    fn(item, (err, result) => {
      if (finished) return;
      if (err) {
        finished = true;
        return callback(err, []);
      }
      results.push(result);
      if (--remaining === 0) callback(null, results);
    })
  });
}
```

```
function throwDiceAndSave(player, callback) {
  retryThrowDice((err, dice1) => {
    if (err) return callback(err);
    retryThrowDice((err, dice2) => {
      if (err) return callback(err);
      const score = { player, score: [dice1, dice2] };
      retrySavePlayerScore(score, callback);
    });
  });
}

retryGetPlayers((err, players) => {
  if (err) return console.log(err);
  asyncMap(throwDiceAndSave, players, (err) => {
    if (err) return console.log(err);
    retryGetScoreBoard(console.log);
  })
});
```

```
function throwDiceAndSave(player, callback) {
  retryThrowDice((err, dice1) => {
    if (err) return callback(err);
    retryThrowDice((err, dice2) => {
      if (err) return callback(err);
      const score = { player, score: [dice1, dice2] };
      retrySavePlayerScore(score, callback);
    });
  });
}
```

```
retryGetPlayers((err, players) => {
  if (err) return console.log(err);
  asyncMap(throwDiceAndSave, players, (err) => {
    if (err) return console.log(err);
    retryGetScoreBoard(console.log);
  })
});
```

```
function throwDiceAndSave(player, callback) {
  retryThrowDice((err, dice1) => {
    if (err) return callback(err);
    retryThrowDice((err, dice2) => {
      if (err) return callback(err);
      const score = { player, score: [dice1, dice2] };
      retrySavePlayerScore(score, callback);
    });
  });
}

retryGetPlayers((err, players) => {
  if (err) return console.log(err);
  asyncMap(throwDiceAndSave, players, (err) => {
    if (err) return console.log(err);
    retryGetScoreBoard(console.log);
  })
});
```

```
function throwDiceAndSave(player, callback) {
  retryThrowDice((err, dice1) => {
    if (err) return callback(err);
    retryThrowDice((err, dice2) => {
      if (err) return callback(err);
      const score = { player, score: [dice1, dice2] };
      retrySavePlayerScore(score, callback);
    });
  });
}

retryGetPlayers((err, players) => {
  if (err) return console.log(err);
  asyncMap(throwDiceAndSave, players, (err) => {
    if (err) return console.log(err);
    retryGetScoreBoard(console.log);
  })
});
```

```
function asyncMapSeries(fn, list, callback) {
  const results = [];
  let listCopy = list.slice();
  const next = () => {
    if (listCopy.length > 0) {
      const element = listCopy.shift();
      fn(element, (err, result) => {
        if (err) return callback(err, []);
        results.push(result);
        next();
      });
    } else {
      callback(null, results);
    }
  };
  next();
}
```

```
function asyncMapSeries(fn, list, callback) {
  const results = [];
  let listCopy = list.slice();
  const next = () => {
    if (listCopy.length > 0) {
      const element = listCopy.shift();
      fn(element, (err, result) => {
        if (err) return callback(err, []);
        results.push(result);
        next();
      });
    } else {
      callback(null, results);
    }
  };
  next();
}
```

```
function asyncMapSeries(fn, list, callback) {
  const results = [];
  let listCopy = list.slice();
  const next = () => {
    if (listCopy.length > 0) {
      const element = listCopy.shift();
      fn(element, (err, result) => {
        if (err) return callback(err, []);
        results.push(result);
        next();
      });
    } else {
      callback(null, results);
    }
  };
  next();
}
```

```
function asyncMapSeries(fn, list, callback) {
  const results = [];
  let listCopy = list.slice();
  const next = () => {
    if (listCopy.length > 0) {
      const element = listCopy.shift();
      fn(element, (err, result) => {
        if (err) return callback(err, []);
        results.push(result);
        next();
      });
    } else {
      callback(null, results);
    }
  };
  next();
}
```

```
function throwDiceAndSave(player, callback) {
  retryThrowDice((err, dice1) => {
    if (err) return callback(err);
    retryThrowDice((err, dice2) => {
      if (err) return callback(err);
      const score = { player, score: [dice1, dice2] };
      retrySavePlayerScore(score, callback);
    });
  });
}

retryGetPlayers((err, players) => {
  if (err) return console.log(err);
  asyncMapSeries(throwDiceAndSave, players, (err) => {
    if (err) return console.log(err);
    retryGetScoreBoard(console.log);
  })
});
```

```
function throwDiceAndSave(player, callback) {
  retryThrowDice((err, dice1) => {
    if (err) return callback(err);
    retryThrowDice((err, dice2) => {
      if (err) return callback(err);
      const score = { player, score: [dice1, dice2] };
      retrySavePlayerScore(score, callback);
    });
  });
}

retryGetPlayers((err, players) => {
  if (err) return console.log(err);
  asyncMapSeries(throwDiceAndSave, players, (err) => {
    if (err) return console.log(err);
    retryGetScoreBoard(console.log);
  })
});
```

Observables

Observables

- Traditionally:
  - decouple objects to avoid dependencies
- Javascript
  - associate multiple callbacks to the same agent
  - model more sophisticated processes

Observables

- One **producer**
- Multiple **consumers**
- Communication with **events**

```
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));
```

Observable

- `on(event, callback)`
  - **event**: string with the name of the event
  - **callback**: function to associate to that event

Observable

- `off(event, callback)`
  - **removes** the association between **event** and **callback**
  - In other words: **callback** won't run anymore when **event** is fired

Observable

- `emit(event, ...args)`
  - **emits** the event **event** with the parameters **args**
  - All **callbacks** associated to **event** are executed with **args**

Exercise: Observable

- Implement the class **`Observable`**
  - With its three methods: **on**, **off** y **emit**
  - To make the previous example work

```
class Observable {
  constructor() {

  }
  on(event, cb) {

  }
  off(event, cb) {

  }
  emit(event, ...payload) {

  }
}
```

Observables

- Probably *the most important pattern in Javascript*
  - In *node.js* is called **EventEmitter**
  - Foundation for more complex patterns
    - Queues
    - Streams
    - ...

Exercise: Observable

- Create a class **Dice** that emits **random values between 1 and 6** emitting the event **“throw”** at random intervals between 100 and 500ms
- Create two instances of **Dice**
- Write the needed code to log **pairs of throws** (the first throw of the first class with the first of the second, etc...)

Promises

Promises

- Higher level abstraction
- **VERY** useful
- **VERY** popular

Promises

- A promise represents **a future value**

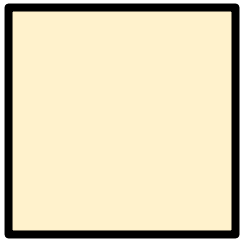
Promises

- A promise is an **object** that acts as a **proxy**
  - between the **producer** of a value
  - and its **consumers**
- To simplify the code dealing with async processes

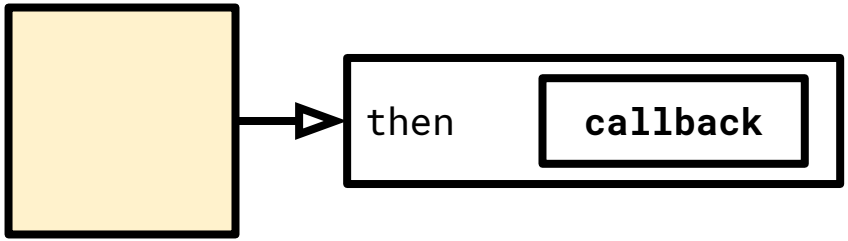
Promises

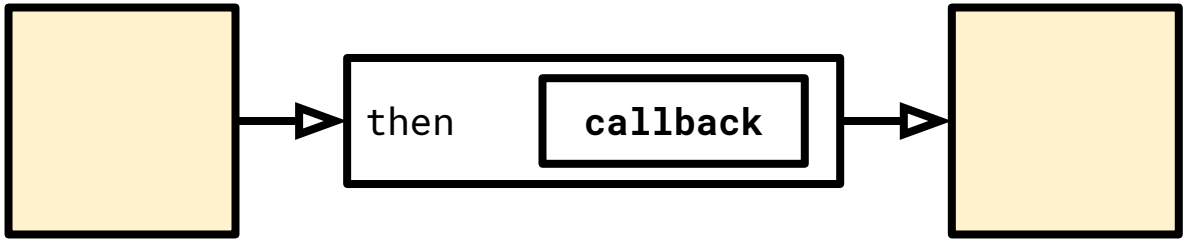
- A promise is just **a box** that **holds** some **value**

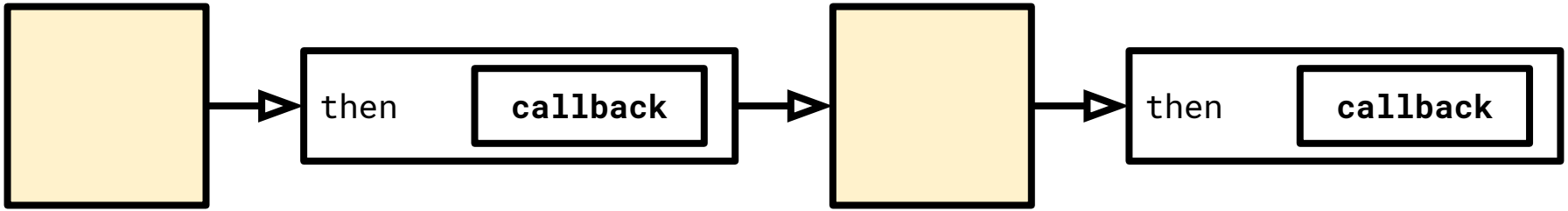
7

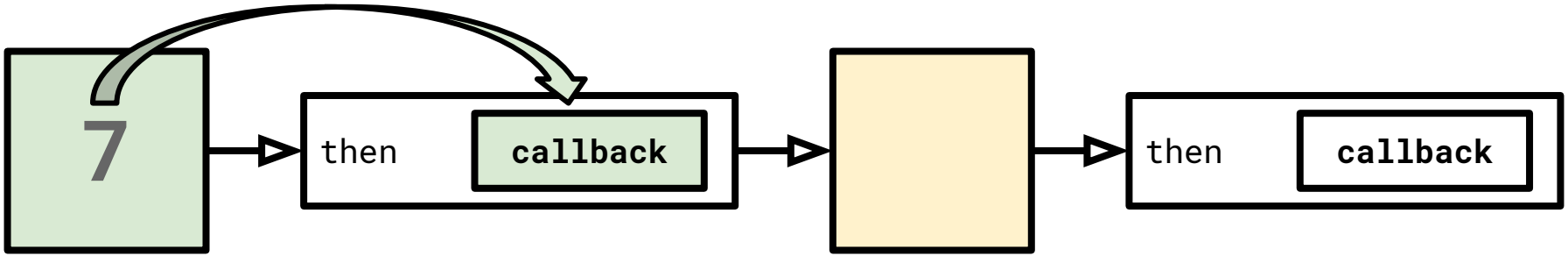


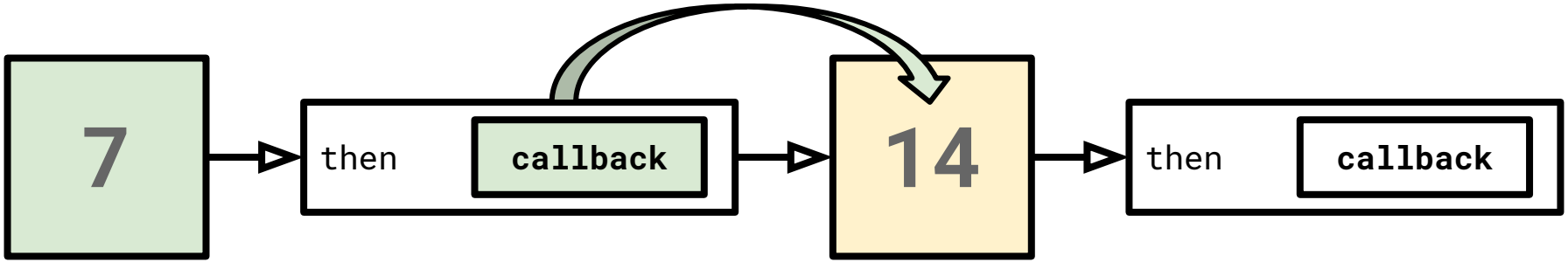
redradix

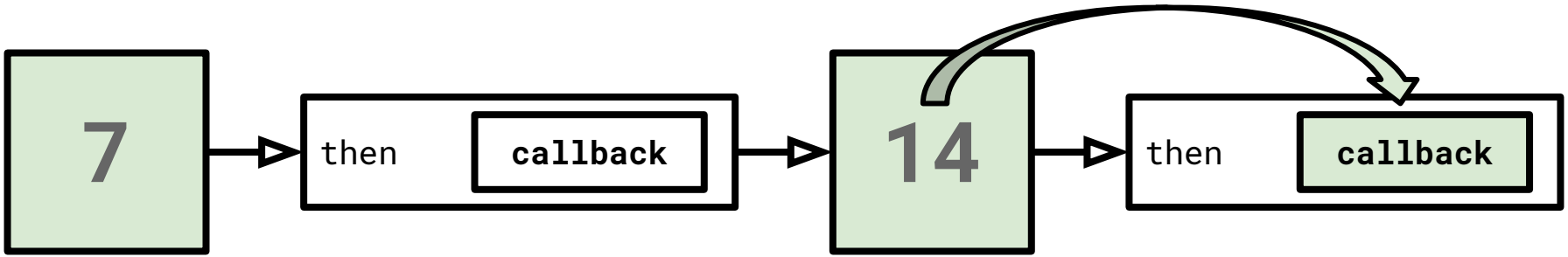






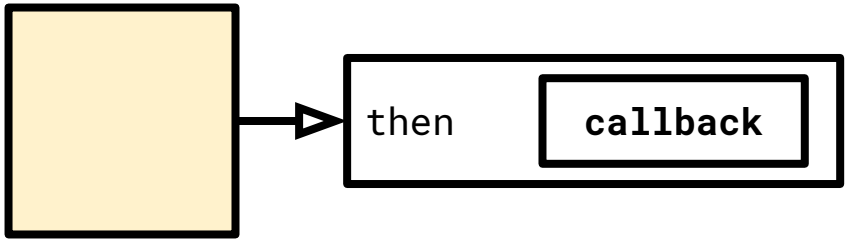


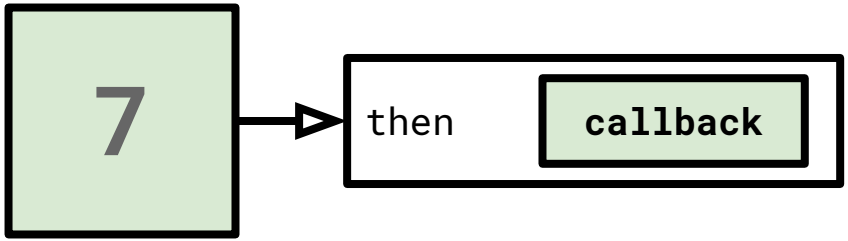


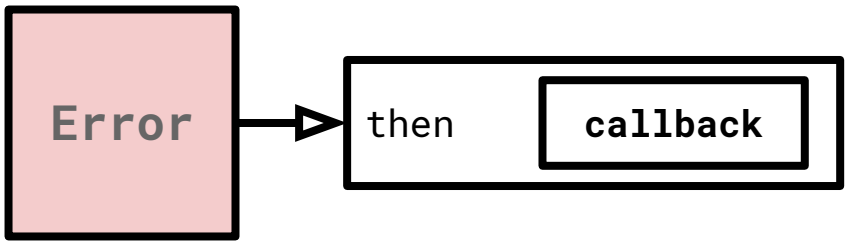


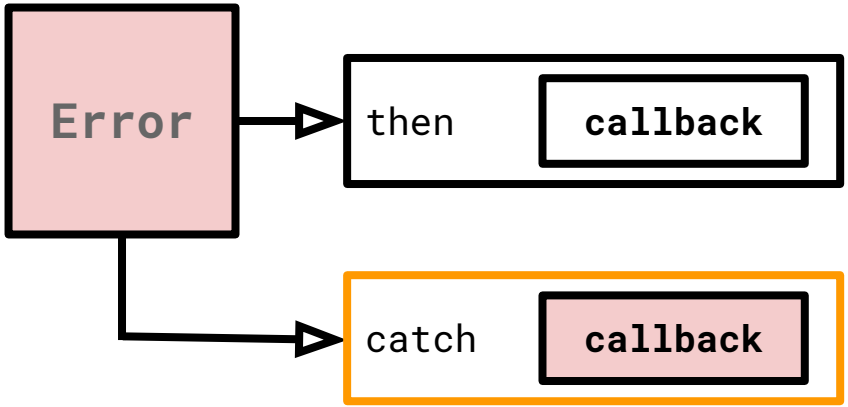
```
const promise2 = promise.then((value) => {
  return value * 2;
});

const promise3 = promise2.then((value2) => {
  console.log(value2);
});
```

```
const promise2 = promise.then((value) => {
  console.log('one');
  return value * 2;
});

console.log('two');

const promise3 = promise2.then((value2) => {
  console.log('three');
  console.log(value2);
});

console.log('four');
```



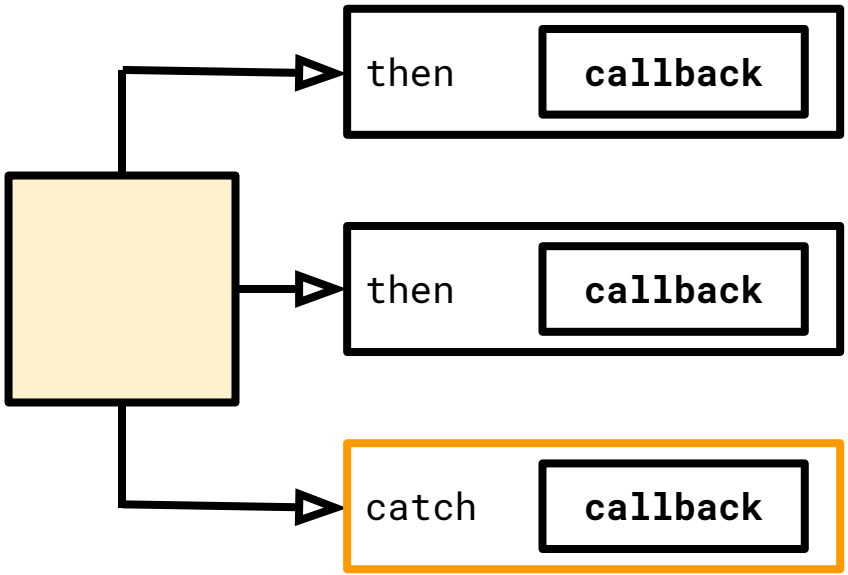


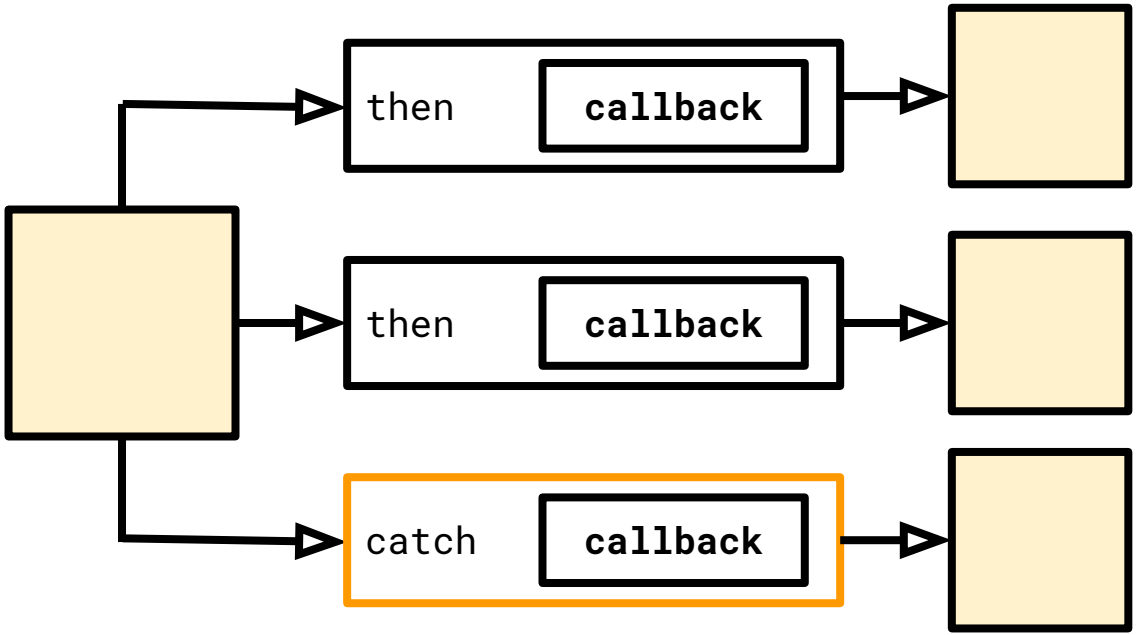


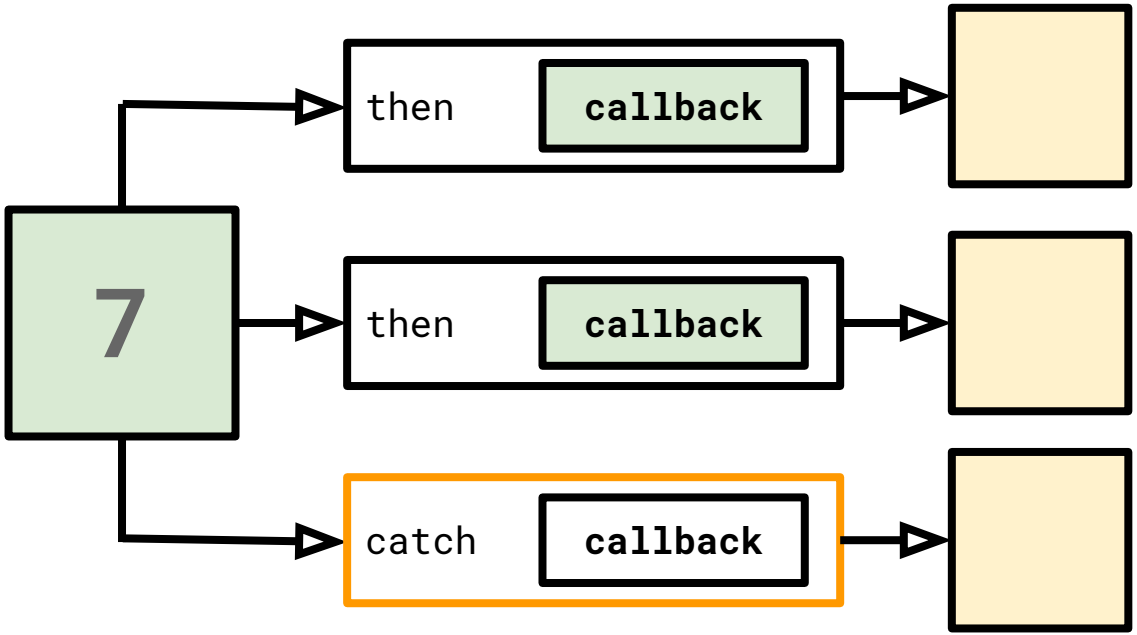


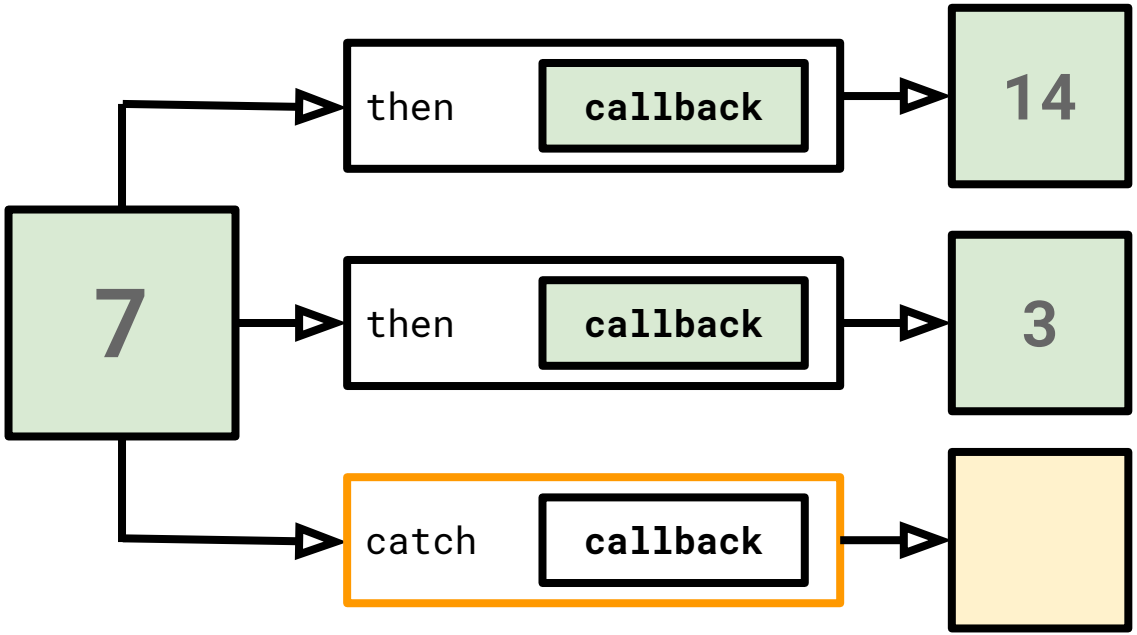
Promesas

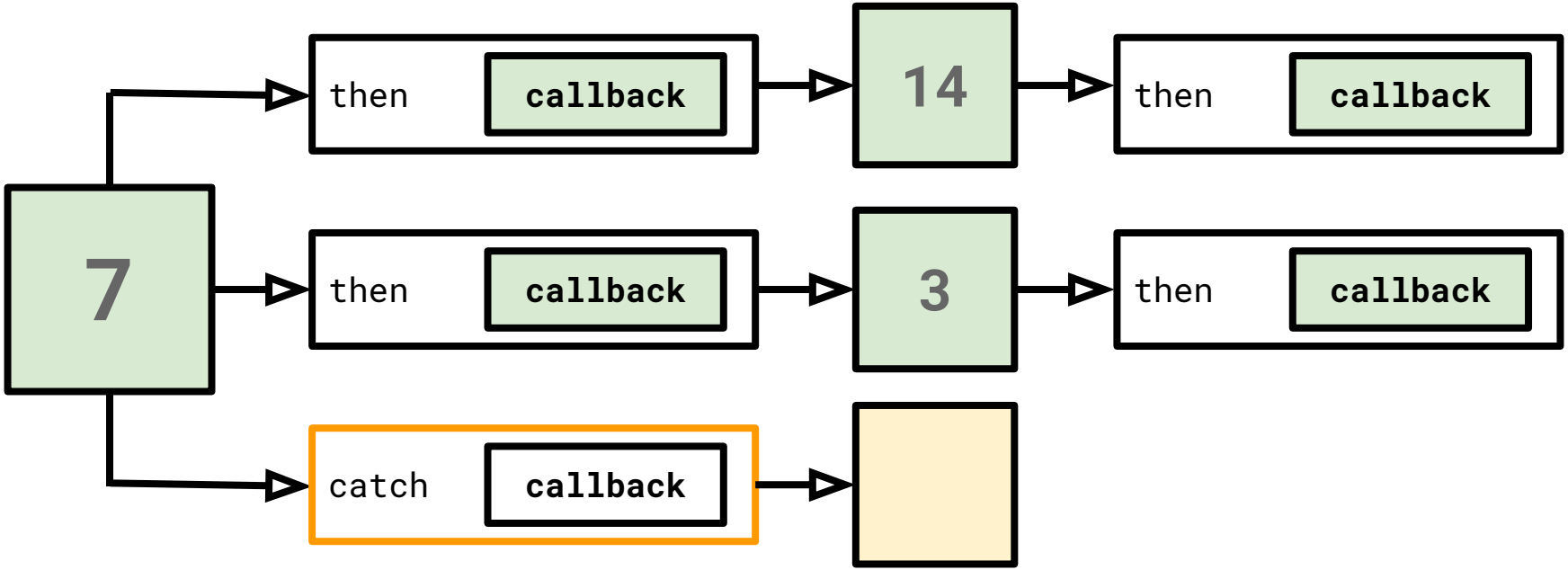
- Promises have three states:
  - pending
  - resolved (or fulfilled)
  - rejected (or failed)
- When a promise is resolved or rejected, **it cannot change its state anymore** and has to stay resolved or rejected forever.





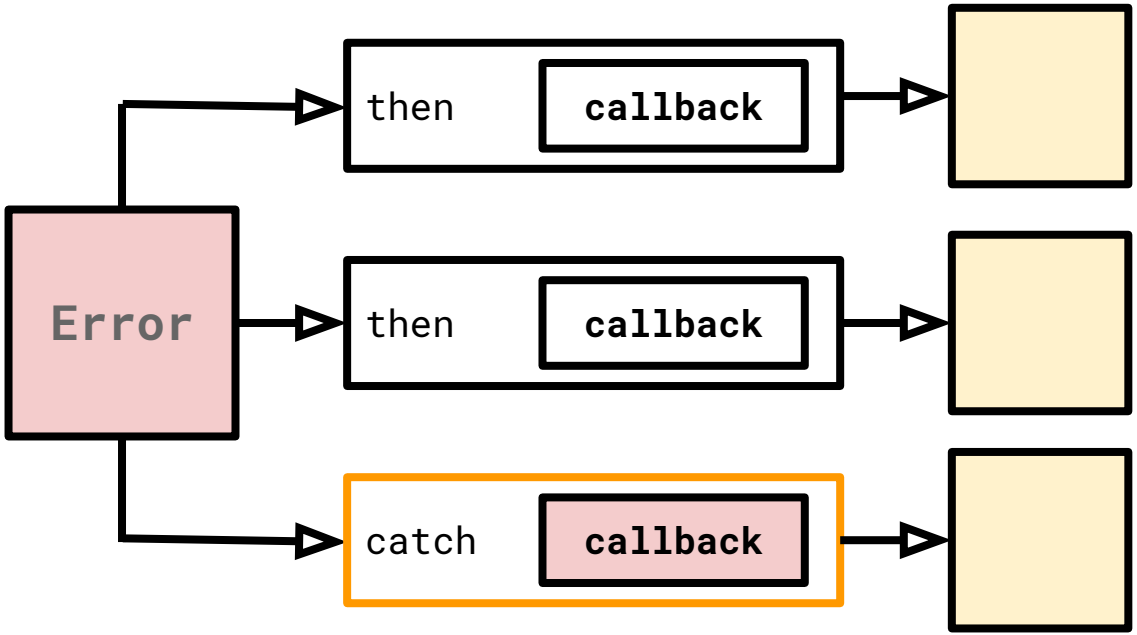


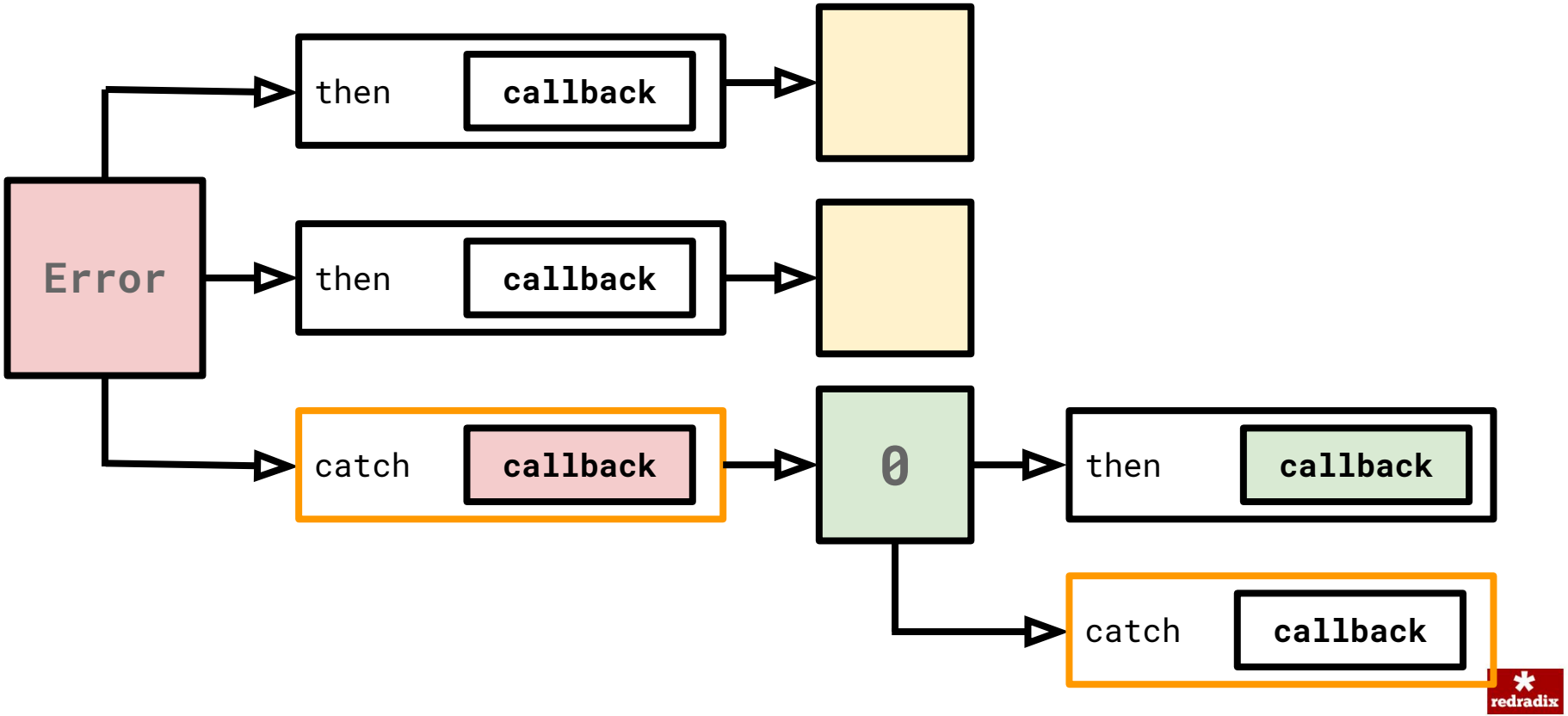

7
then
callback
14
then
callback
3
catch
callback


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

Promises

- Calls to **.then()** and **.catch()** return **a new promise**
- Representing the **return value** of their **callbacks**
- The callback passed to **.then()** runs when (if) the promise gets **resolved**
- The callback passed to **.catch()** runs when (if) the promise gets **rejected**




then
callback
Error
then
callback
catch
callback
0
then
callback
catch
callback
redradix

Promises

- There are **three** ways to create a promise:
  - `Promise.resolve(value)`
  - `Promise.reject(error)`
  - `new Promise(...)`

Promises

- `Promise.resolve(value)`
  - Creates a **resolved promise**
  - **callbacks** passed to **.then** run **immediately**

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

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

Promises

- `Promise.reject(error)`
  - Creates a **rejected promise**
  - **callbacks** passed to **.catch** run **immediately**

```
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);
```

Promises

- `new Promise(callback)`
  - Creates a **pending promise**
  - Callback runs with **delay 0**
  - **callback** gets two parameters:
    - **resolve**: function to resolve the promise
    - **reject**: function to reject the promise

```
const p = new Promise((resolve, reject) => {
  if (Math.random() < 0.5) resolve('good to go!');
  else reject(new Error('bad luck'));
});

p.then(console.log);
p.catch(console.log);

console.log('before or after?')
```

```
const p = new Promise((resolve, reject) => {
  setTimeout(() => {
    if (Math.random() < 0.5) resolve('good to go!');
    else reject(new Error('bad luck'));
  }, 1000);
});

p.then(console.log);
p.catch(console.log);

console.log('before or after?')
```

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

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

getDate((date) => {
  // continue here
});

// ???
```

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

getDate((date) => {
  getDate((date2) => {
    // continue here
  });
  // ???
});
```

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

getDate((date) => {
  getDate((date2) => {
    getDate((date3) => {
      // continue here
    });
    // ???
  });
});
```

```
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();
// continue here
```

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

const datePromise = getDate();
const datePromise2 = getDate();
const datePromise3 = getDate();
// continue here
```

Promises

- `.then(resolveCallback, [rejectCallback])`
  - Receives one (or two) callbacks
  - **resolveCallback**: receives the resolution value
  - **rejectCallback**: receives the rejection error

```
const p = Promise.resolve(true);

p.then(
  v => { throw new Error('rejected!'); },
  err => console.error(err)
);
```

Promises

- `.then(resolveCallback, [rejectCallback])`
  - Returns **a new promise**
    - **resolved** with the value returned by **resolveCallback** or **rejectCallback**
    - it's **rejected** if the callback **throws an exception**

```
const p1 = new Promise(
  resolve => setTimeout(() => resolve(1), 1000)
);

const p2 = p1.then(v => v + 1);
const p3 = p2.then(v => v + 1);
const p4 = p3.then(v => v + 1);

console.log(p4); // ???
```

```
const p1 = new Promise(
  resolve => setTimeout(() => resolve(1), 1000)
);

const p2 = p1.then(v => v + 1);
const p3 = p2.then(v => v + 1);
const p4 = p3.then(v => v + 1);

p4.then(console.log); // ???
```

```
const p1 = new Promise(
  resolve => setTimeout(() => resolve(1), 1000)
);

const p2 = p1.then(v => v + 1);
const p3 = p2.then(v => v + 1);
const p4 = p3.then(v => v + 1);

p4.then(console.log); // ???

console.log('before or after?');
```

```
const p1 = new Promise(
  resolve => setTimeout(() => resolve(1), 1000)
);

p1.then(v => v + 1)
  .then(v => v + 1)
  .then(v => v + 1)
  .then(console.log);
```

```
const p1 = new Promise(
  resolve => setTimeout(() => resolve(1), 1000)
);

p1.then(v => v + 1)
  .then(v => v + 1)
  .then(v => v + 1)
  .then(console.log);
```

Promises

- If we have a promise **A**
- Calling **A.then(callback)** gives a new promise **B**
- The promise **B** will **be resolved** with the value **returned by the callback**

```
const b = a.then(() => 1);
b.then(console.log);
```

Promises

- If we have a promise **A**
- Calling **A.then(callback)** gives a new promise **B**
- The promise **B** will **be resolved** with the value **returned by the callback**
- But... **What if the callback returns another promise C?**

```
const b = a.then(() => {
  const c = new Promise(
    resolve => setTimeout(() => resolve(1), 1000)
  );
  return c;
});

b.then(console.log);
```

Promises

- In that case, **B** becomes **a reflection of C**
  - **B** stays *pending* as long as **C** stays *pending*
  - If **C** is resolved, **B** is resolved **with the same value**
  - If **C** is rejected, **B** is rejected **with the same error**

```
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); // ???
```

Promises

- `.then(...)`
  - Create *sequences of async operations*
  - Keeping a **clean execution flow**
  - Without having to *indent* each step

Promises

- `.catch(rejectCallback)`
  - equivalent to `.then(identity, rejectCallback)`
  - allow us to **capture the rejection error**

```
const p1 = new Promise((resolve, reject) => {
  setTimeout(() => reject(new Error('Panic!')), 100);
});

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

```
const p1 = new Promise((resolve, reject) => {
  setTimeout(() => reject('Panic!'), 100);
});

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

Promises

- A promise is considered **rejected** if **an exception is thrown** during the execution of...
  - ...its *resolve callback*
  - ...its *rejection callback*
  - ...the *callback* passed to **new Promise**(...)

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

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

Promises

- `.catch(rejectCallback)`
  - Returns **a promise**
  - Behaves exactly like the promise returned by **.then()**
  - The **resolution** value is the value returned from **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!'),
    () => 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'));
```

Promises

- If we have a promise **A**
- And calling **A.then(callback)** returns the promise **B**
- The promise **B** will be **resolved** with the value returned by the callback

Promises

- If we have a promise **A**
- And calling **A.then(callback)** returns the promise **B**
- The promise **B** will be **resolved** with the value returned by the callback
- **What happens if A is rejected and we haven't specified any** *rejectCallback* **in .then(...)?**

Promises

- In that case, **B** is **rejected** with the same error as **A**
- In other words, **rejection propagates downwards**

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

p1
  .then(() => console.log('1...')) // no rejectCallback -> rejected!
  .then(() => console.log('2...')) // no rejectCallback -> rejected!
  .then(() => console.log('3...')) // no rejectCallback -> rejected!
  .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'));
```

Promises

- In a chain of promises
  - Errors **propagate downwards**
  - If the error is captured, the execution goes back to normal
- Makes async error handling much easier

Promises

- `Promise.all([prom1, prom2, prom3, …])`
  - Returns **a new promise**
  - Resolved when **every promise is resolved**
  - **Resolution value**: array with resolution values of each promise
  - If **one promise is rejected**, the result is **rejected too, with the same 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
```

Promises

- `Promise.race([prom1, prom2, prom3, …])`
  - Returns **a new promise**
  - **Reflects** the resolution or rejection value of **the first promise being resolved or rejected**

```
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
```

Exercise: Promises

- Implement *mapPromise(fn, promisesOrValues)*
  - Apply **fn** to the *resolution value* of each promise of the list
  - In parallel
  - Returns a **new promise**
  - That resolves to the **list of results**

Exercise: Promises

- Implement *mapSeriesPromise(fn, promisesOrValues)*
  - Similar to *mapPromise*
  - But **the iteration happens serially**

Exercise: Promises

- Implement *reducePromise(fn, init, promisesOrValues)*
  - Similar to *reduce*, but with promises
  - **Iteration in series**

Exercise: Promises

```
reducePromise(
  (acc, el) => futureValue(acc + i, 100),
  [0, 1, 2, 3, futureValue(4), 5, 6, 7, 8, 9],
  0
)
  .then(console.log); // 45
```

Coroutines

Coroutines

- *generators* have a unique power:
  - **stop the execution flow** at any point
  - and being able to **resume it later from the same point**
  - without using 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);
```

Coroutines

- How could we take advantage of this?
  - stop the execution every time we call an asynchronous function...
  - ...and resume it when the function has finished?

Coroutines

1. Write the consumer code in a generator
2. From there, **yield promises**
3. When the promise **is resolved**, we continue the execution
4. If the promise is **rejected**, throw **an exception**

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

function* () {
  console.log('wait until the value is ready…');
  const value = yield futureValue(10, 1000);
  console.log('the value is:', value);
  const double = yield futureValue(value * 2, 1000);
  console.log('double it and you will get:', double);
}
```

Coroutines

We need a function that:

1. Gets the generator with our code
2. Calls it to create an iterator
3. Calls **.next()**
4. Retrieves the **yielded promise** and then waits until...
    a. the promise is **rejected**, and then throws an exception
    b. the promise is **resolved**, and then resumes the execution
5. GOTO 3

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

const fn = co(function* () {
  console.log('wait until the value is ready…');
  const value = yield futureValue(10, 1000);
  console.log('the value is:', value);
  const double = yield futureValue(value * 2, 1000);
  console.log('double it and you will get:', double);
});

fn();
```

Exercise: Corrutines

- Implement the function **co(generator)**
  - Receives a *generator*
  - Returns a function
  - Calling the function will run the generator, advancing the execution and waiting on the yielded promises, until the generator finishes

Coroutines

- We can capture errors with **try/catch**

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

Coroutines

- We can use **yield** in every expression

```
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']
})();
```

Coroutines

- We can use every promise combination method

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

Coroutines

- The **return value** of the coroutine is **a promise**
  - If the generator throws an exception, the promise gets **rejected** with the thrown error
  - If the generator reaches its end, the promise will be **resolved** with the last returned value

```
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
});
```

Coroutines

- ES2017 introduces **native coroutines**
  - **async** to declare a function as a coroutine
  - **await** to wait for the resolution of promises

```
(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);
})();
```