



UNIVERSIDADE
FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Promises

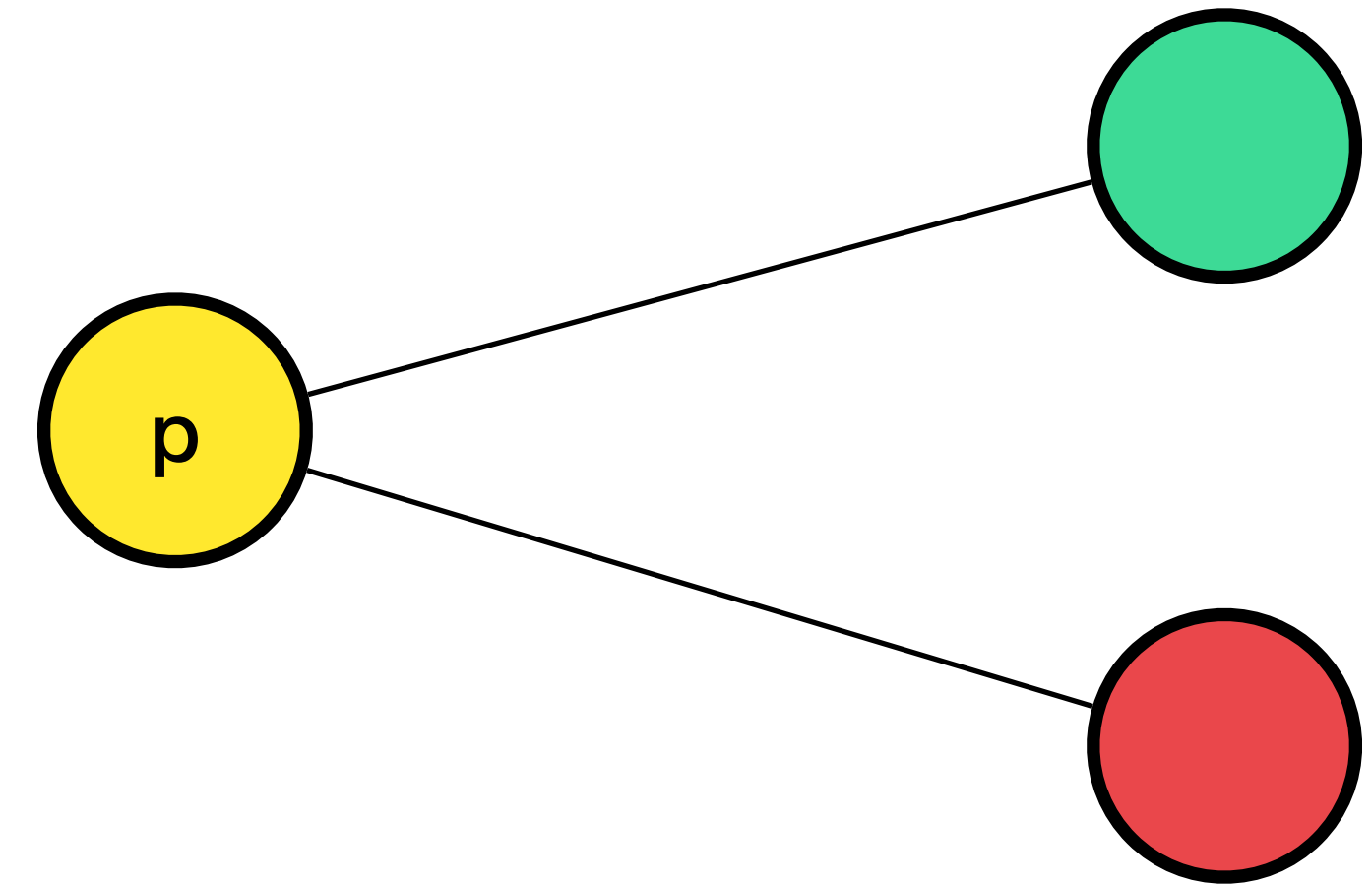
QXD0020 - Desenvolvimento de Software para Web

Prof. Bruno Góis Mateus (brunomateus@ufc.br)

Agenda

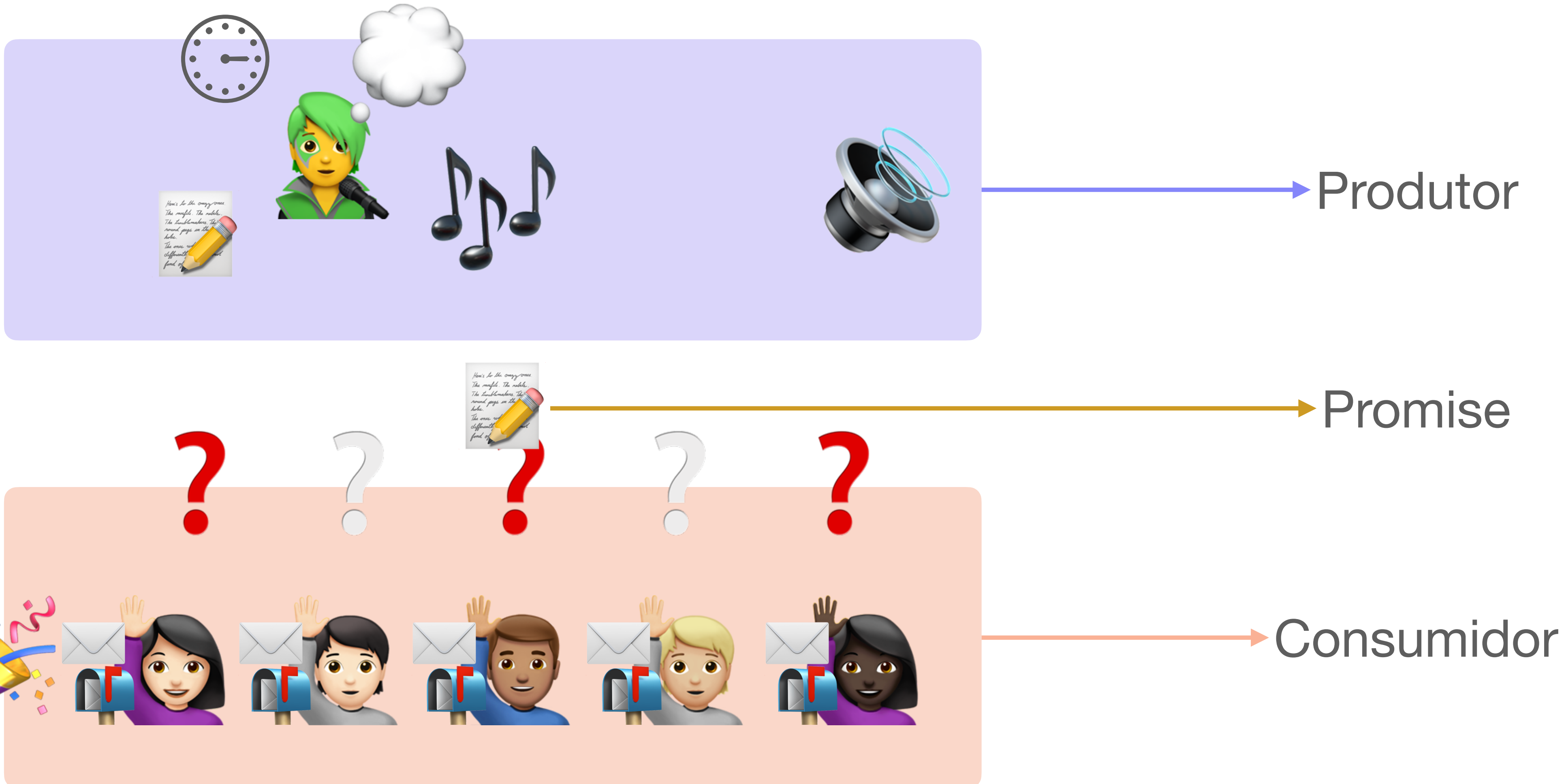
- Introdução
- Estados e consumidores
- Encadeamento
- `sync / await`

Introdução



Promises

Introdução



Promises

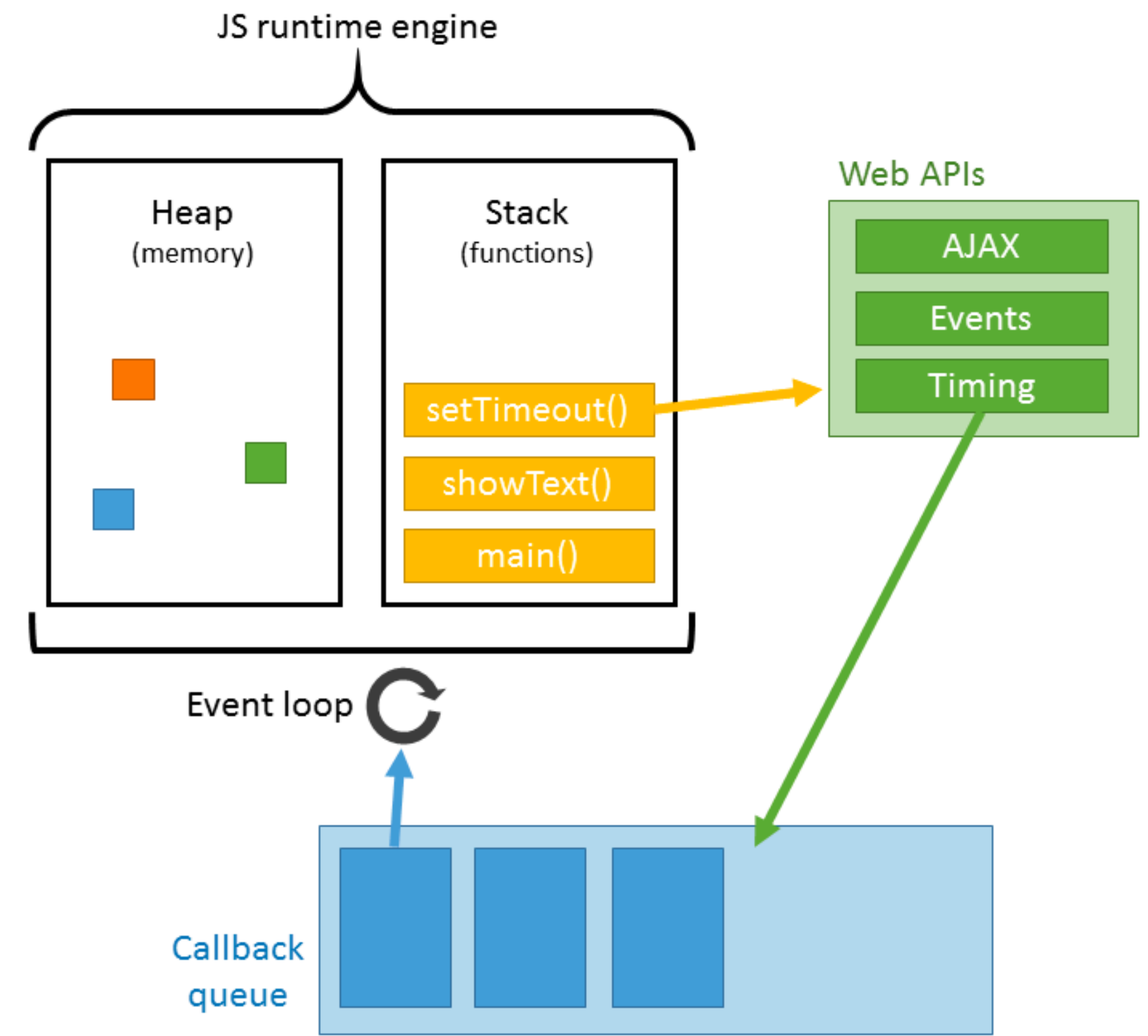
Introdução

- O conceito existe desde à década de 70, quando eram chamadas de futures, deferred ou delays
- Em JavaScript surgiram inicialmente em 2007 na biblioteca MochiKit
- Com o lançamento do ES6, passaram a existir nativamente
- Definição:
 - Um objeto que representa a eventual completude ou falha de uma operação assíncrona

Promises

Motivação

- JavaScript é uma linguagem single thread
 - Não é possível realizar mais uma operação ao mesmo tempo
 - No navegador ela compartilha um thread com diversas funções do próprio navegador

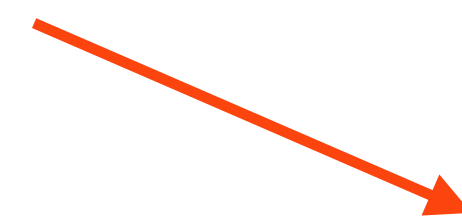


Representação das funções internas de um navegador [Fig. 2]

Promises

Motivação


```
function callbackDeSucesso(result) {  
  console.log(`Pagamento realizado com sucesso`)  
}  
  
function callbackDeFalha(error) {  
  console.error(`Não foi possível realizar o pagamento`)  
}  
  
createAudioFileAsync(audioSettings).then(successCallback, failureCallback);  
pagarComCartao(valor, callbackDeSucesso, callbackDeFalha)  
  
pagarComCartao(valor).then(callbackDeSucesso, callbackDeFalha)
```



Promise

Promises

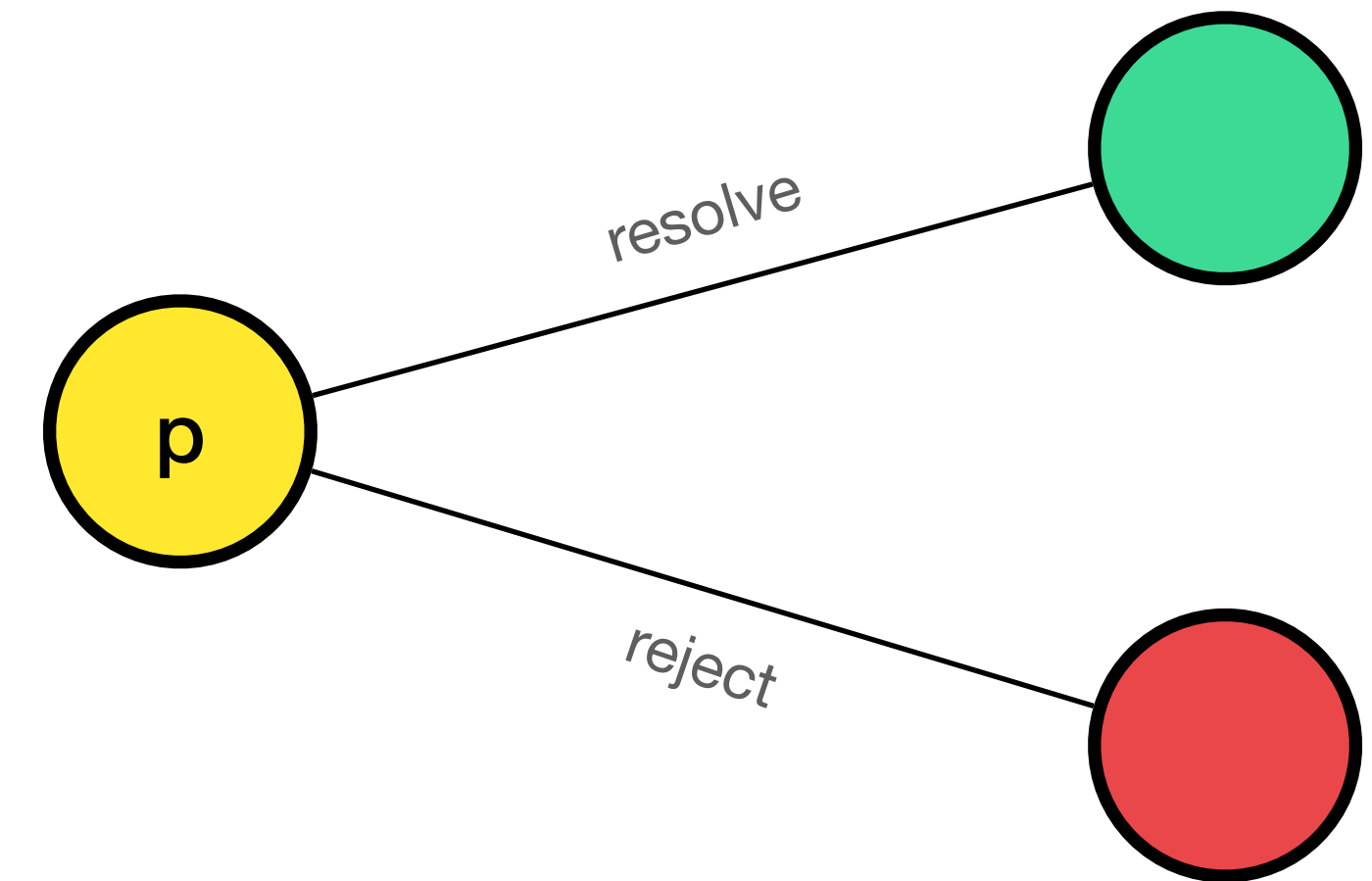
Motivação



```
function pagarComCartao(valor, cartao, function (result) {
  verificarCartao(cartao, function (cartaoValido) {
    verificarSaldo(cartaoValido, function (cartaoValor, valor) {
      verificarFraude(cartaoValido, valor, function (finalResult) {
        pagar(valor, cartao, valor, function (finalResult) {
          console.log(`Pagamento realizador com sucesso`);
        }, failureCallback);
      }, failureCallback);
    }, failureCallback);
  }, failureCallback);
}, failureCallback);
```

```
function pagarComCartao(valor, cartao) {
  verificarCartao(cartao)
    .then((cartaoValido) => verificarSaldo(cartaoValido))
    .then((saldoAtual) => verificarFraude(cartao, valor))
    .then((saldoAtual) => pagar(valor, cartao))
    .then((finalResult) => {
      console.log(`Pagamento realizado com sucesso`);
    })
    .catch(failureCallback);
}
```

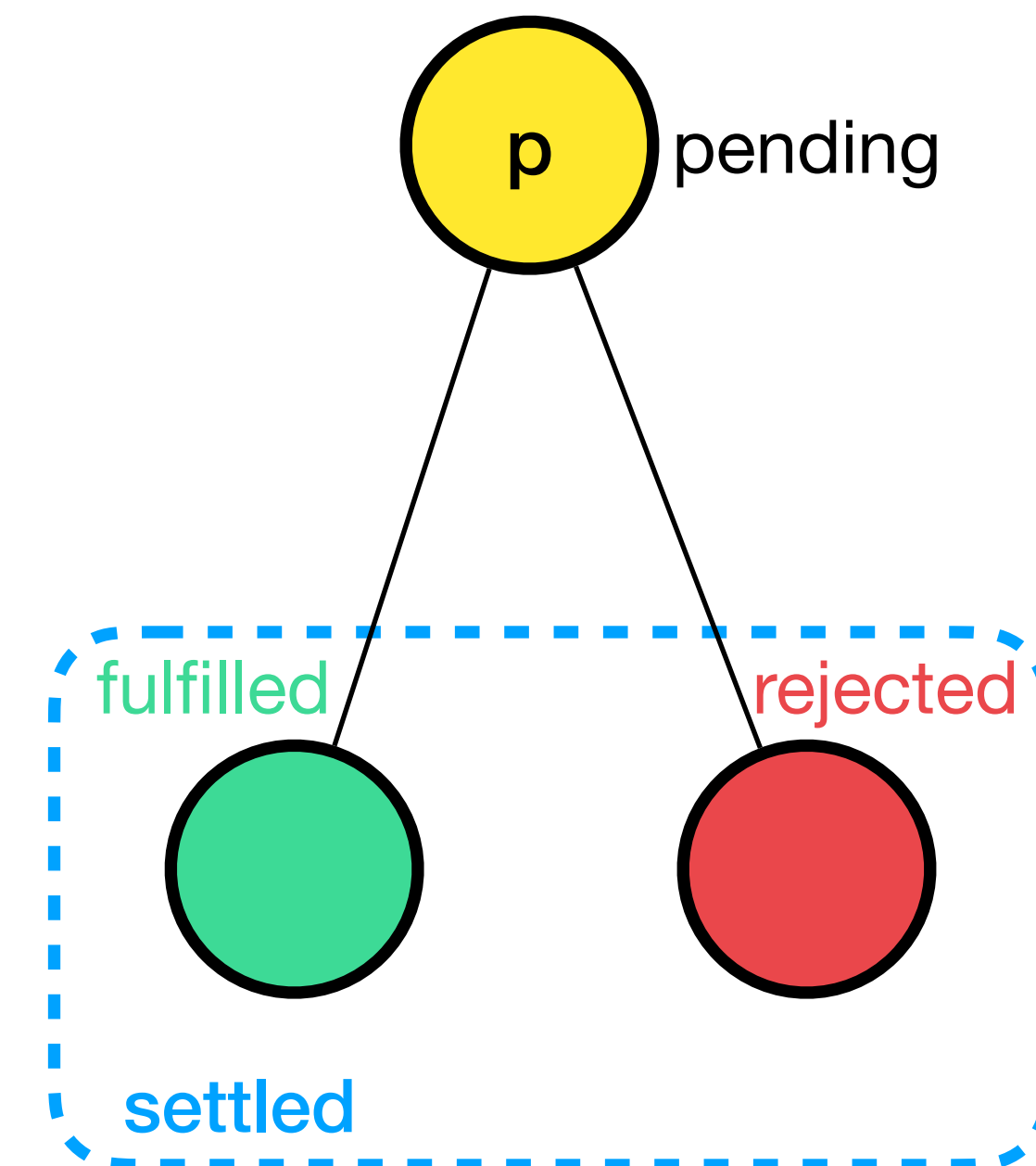

Estados e consumidores



Estados e consumidores

Introdução

- Antes de aprendermos a consumir Promises, vamos entender o seu funcionamento interno
- Uma Promise pode assumir quatro estados
 - pending
 - Estado inicial, nesse estado a promise não foi rejeitada e nem resolvida
 - fulfilled
 - Indica que a operação foi realizada com sucesso, i.e, a promise resolvida
 - rejected
 - Indica que a operação falou, i.e, a promise foi rejeitada
 - settled
 - É o estado final. Indica que a promise foi resolvida ou rejeitada



Estados e consumidores

Consumidores

- Uma promise funciona como uma ligação entre o código produtor e o consumidor
 - Registramos consumidores por dos métodos then e catch
- O then é o mais importante e essencial para o entendimento de promises
- O catch, especialmente útil para casos de falha
- Tanto o then quanto o catch, precisam retornar algum valor
 - Caso contrário as callbacks não terão acesso ao resultado da promise anterior

Estados e consumidores

Consumidores na prática

1. Uso geral:

- Dois parâmetros

```
promise.then(r => console.log(r), e => console.log(e)) 1  
promise.then(r => console.log(r)) 2  
promise.then(null, e => console.log(e)) 3  
promise.catch(e => console.log(e)) 4
```

- Função chamada quando a promise é resolvida
- Função chamada quando a promise é rejeitada

2. Se houver interesse somente nos caso de sucesso podemos usá-lo assim

3. Se houver interesse somente nos caso de falha podemos usá-lo assim

4. Que por sua vez pode ser rescrito da seguinte forma

Estados e consumidores

Finally

- Adicionado no ES9, lançado em 2018
 - Não é usado para registrar um código consumidor
 - Sempre é executado
- Geralmente é utilizado para “limpeza” após a execução da operação
 - Ex: Para indicadores de carregamento, liberar recursos não mais necessários
- Evita a duplicação de código dentro de blocos then e catch

Estados e consumidores

finally na prática

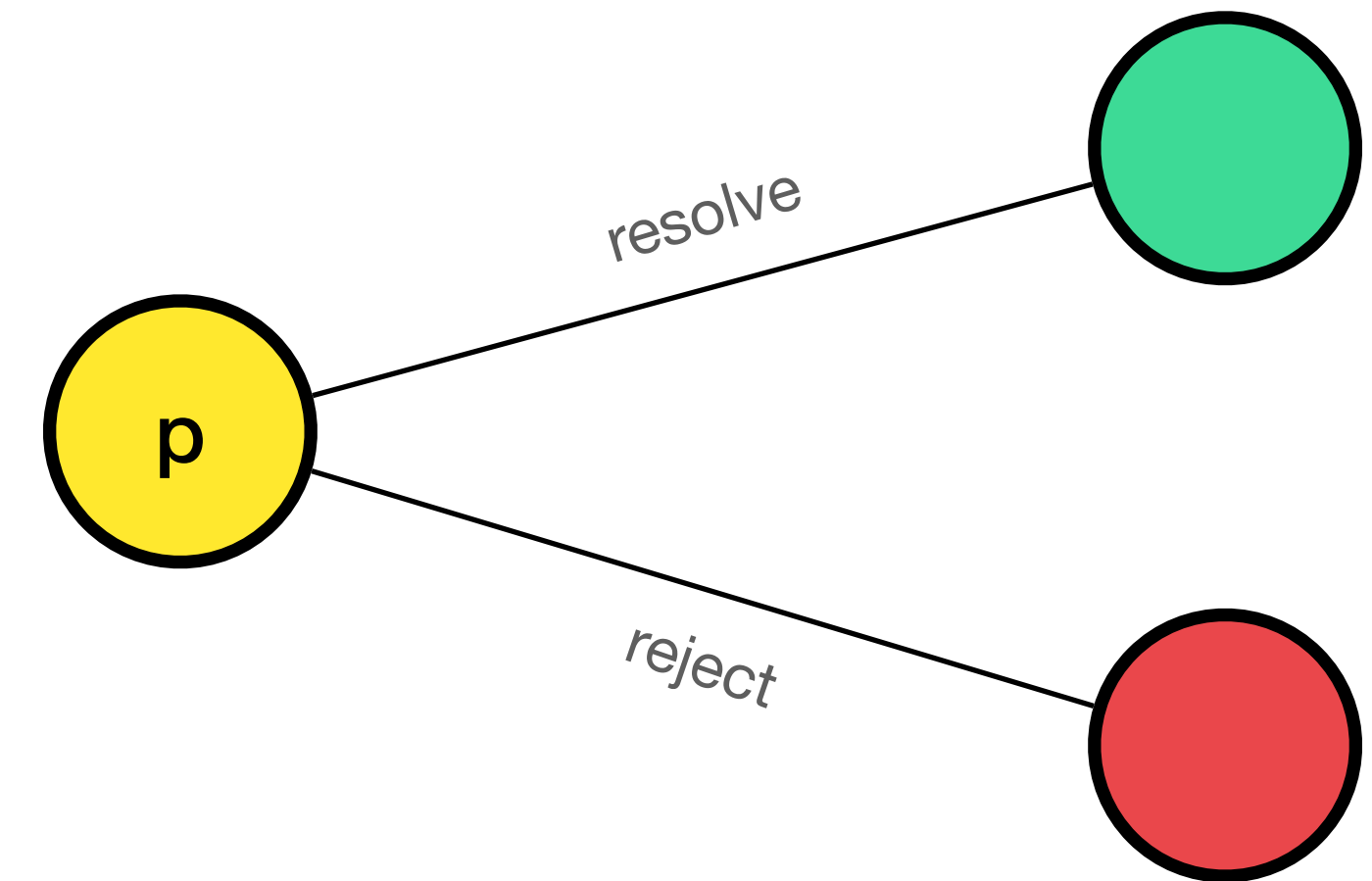
```
let loading = true
const x = setInterval(() => {
  console.log(loading); if(!loading) clearInterval(x)
}, 100)
const promise = new Promise((resolve, reject) => {
  setTimeout(() => resolve("Deu bom"), 300)
})
promise
  .then(r => console.log(r))
  .catch(r => console.log("Deu ruim"))
  .finally(() => { loading = false; console.log("limpando") })
```

```
true
true
true
limpando
Deu bom
false
```

- Não possui argumento. É executado independente resultado da operação
- O resultado da promise ignora o finally e é dirigido ao próximo consumidor
 - O finally não foi criado para processar o resultado das promises

Os três métodos apresentados, `then`, `catch`, `finally`, retornam uma promise, permitindo o encadeamento de promises.

Encadeamento



Encadeamento

Introdução

- Durante o desenvolvimento de uma aplicação web é comum encontrar cenários onde uma requisição assíncrona dependa do resultado de outra
 - Geram códigos complexos ao serem solucionados com uso de callbacks
 - Usando o encadeamento de promises conseguimos algo bem melhor

```
function pagarComCartao(valor, cartao, function (result) {
  verificarCartao(cartao, (cartaoValido) => {
    verificarSaldo(cartaoValido, (cartaoValor, valor) => {
      verificarFraude(cartao, valor, (finalResult) => {
        pagar(cartao, valor, (finalResult) => {
          console.log(`Pagamento realizador com sucesso`);
        }, failureCallback);
      }, failureCallback);
    }, failureCallback);
  }, failureCallback);
}, failureCallback);
```

```
function pagarComCartao(valor, cartao) {
  verificarCartao(cartao)
    .then((cartaoValido) => verificarSaldo(cartaoValido))
    .then((saldoAtual) => verificarFraude(cartao, valor))
    .then((saldoAtual) => pagar(valor, cartao))
    .then((finalResult) => {
      console.log(`Pagamento realizado com sucesso`);
    })
    .catch(failureCallback);
}
```

Encadeamento

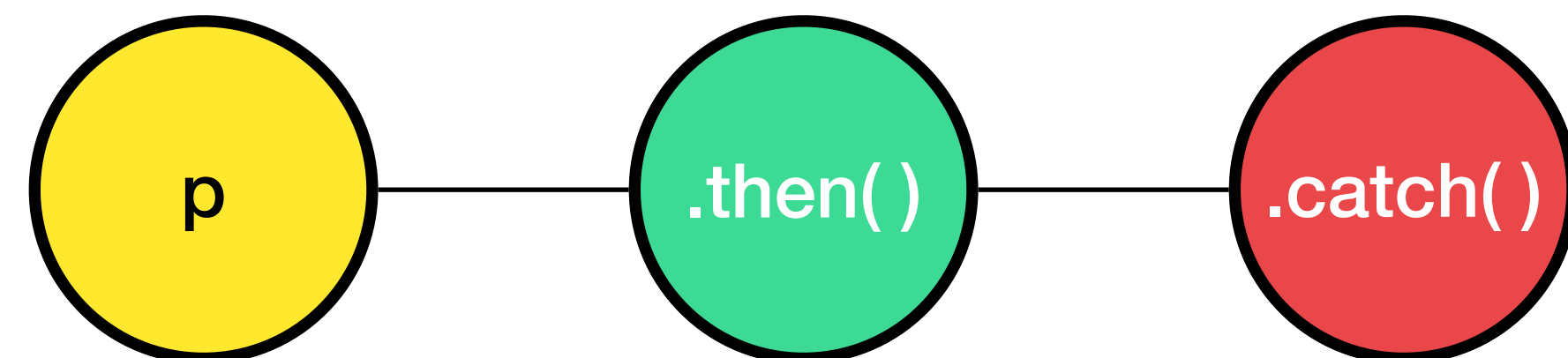
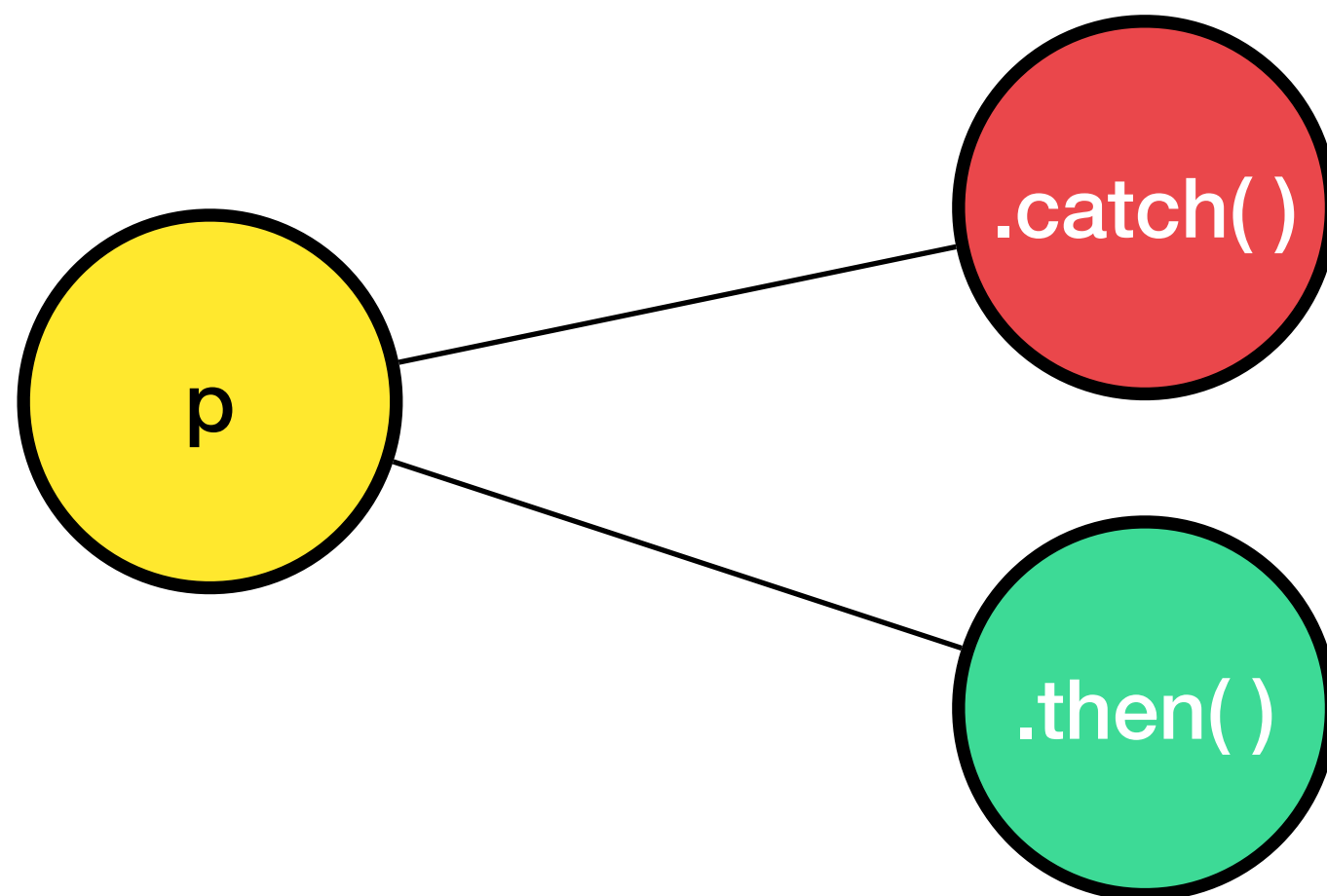
Exemplo prático

```
const p = new Promise((resolve, reject) => {  
  Math.random() > 0.5 ? resolve('yay') : reject('no')  
})
```

```
p.then((res) => {})  
p.catch((rej) => {})
```

```
p.then((res) => {}).catch((rej) => {})
```

- Cenário 1 - Dois *bindings* na mesma promise
- Cenário 2 - Encadeamento entre o then e catch
- O catch será feito na promise retornada pelo then

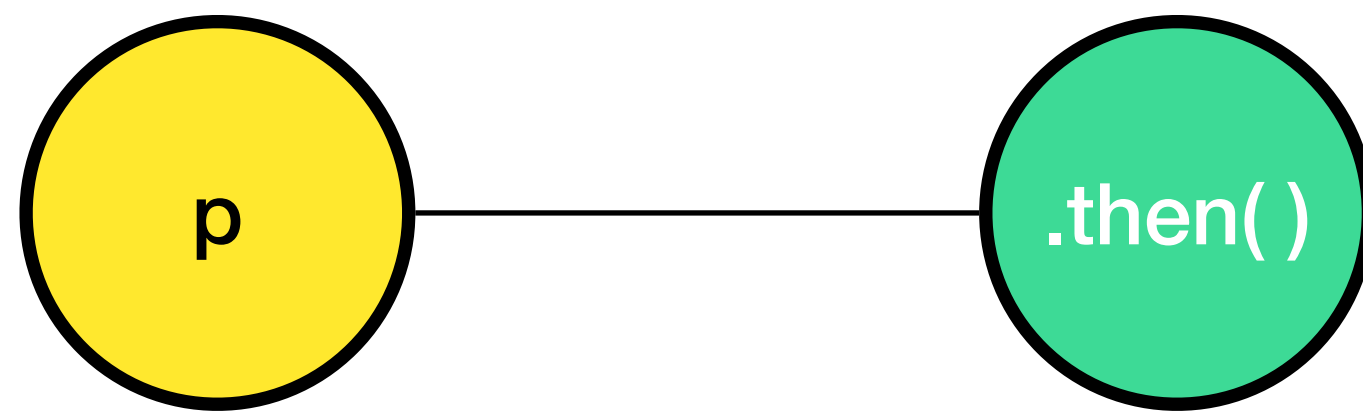


Encadeamento

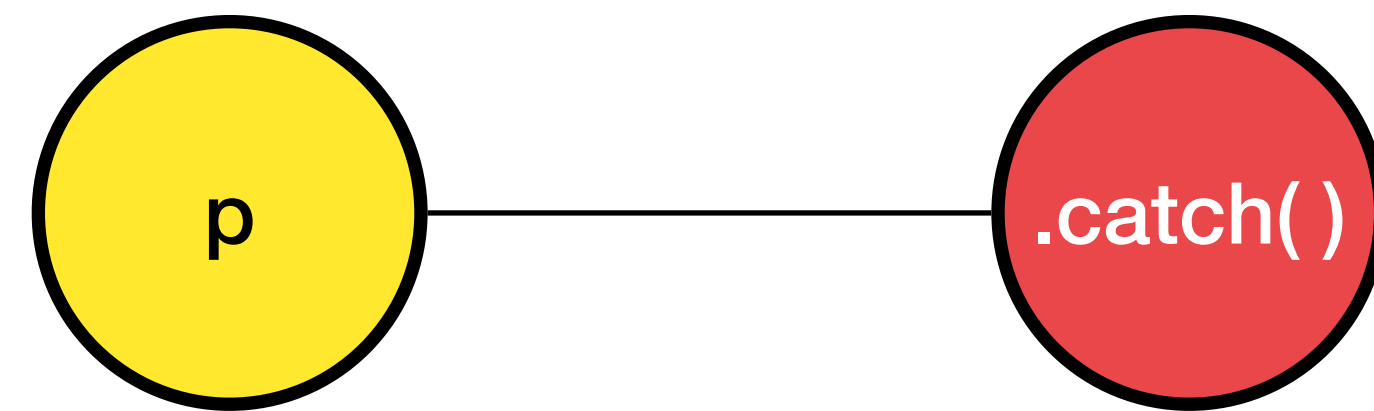
Exemplo prático

```
const p = new Promise((resolve, reject) => {  
  Math.random() > 0.5 ? resolve('yay') : reject('no')  
})  
p.then((res) => {})  
p.catch((rej) => {})
```

Math.random > 0.5



Math.random <= 0.5

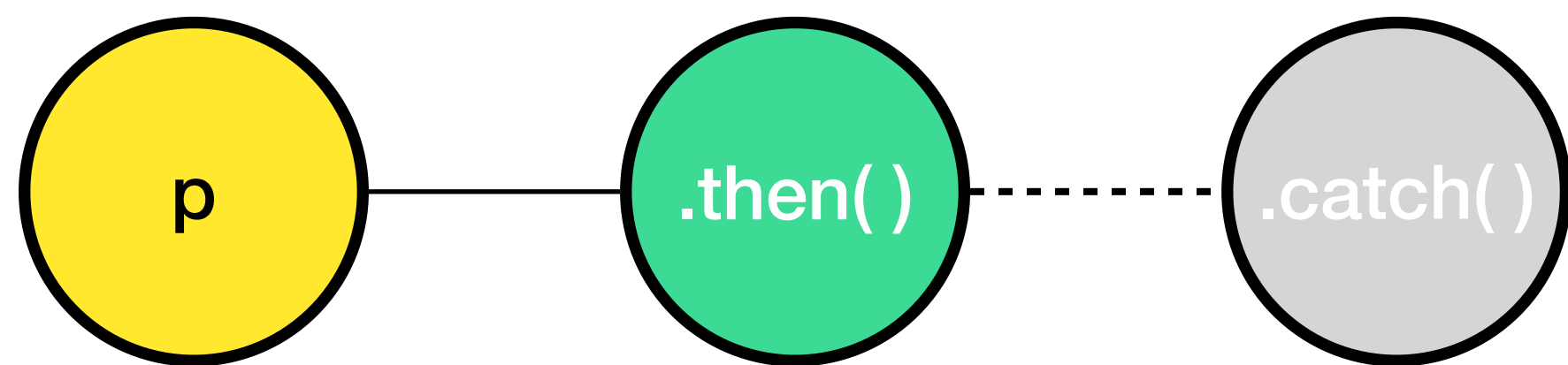


Encadeamento

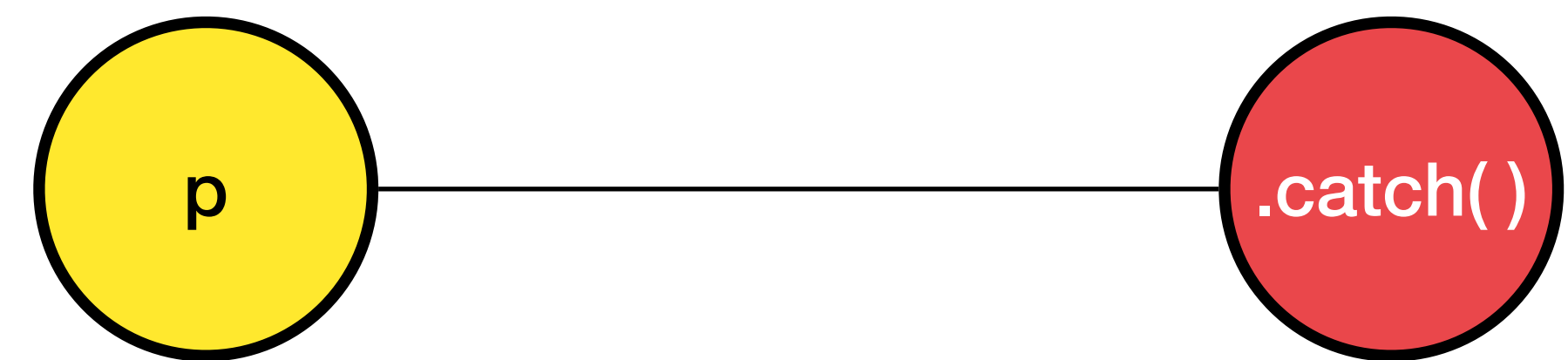
Exemplo prático

```
const p = new Promise((resolve, reject) => {  
  return (Math.random() > 0.5) ? resolve('yay') : reject('no')  
})  
p.then((res) => {}).catch((rej) => {})
```

Math.random > 0.5



Math.random <= 0.5

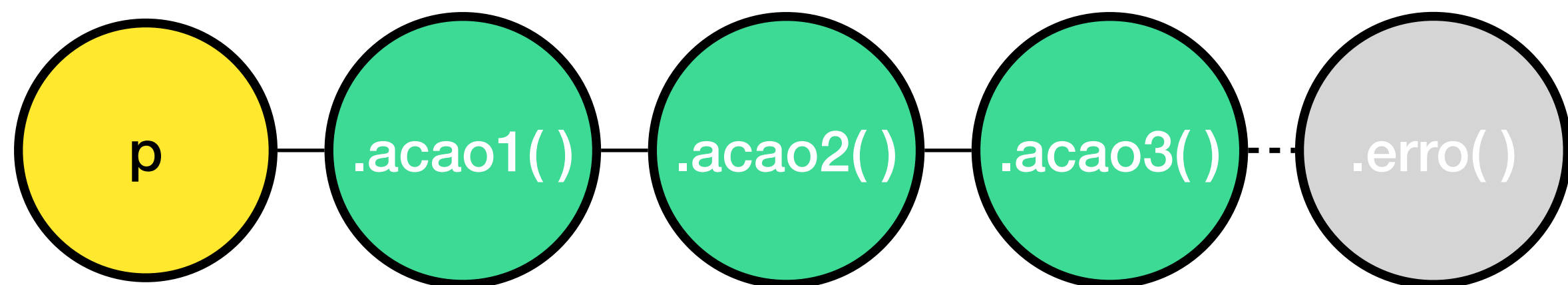


Encadeamento

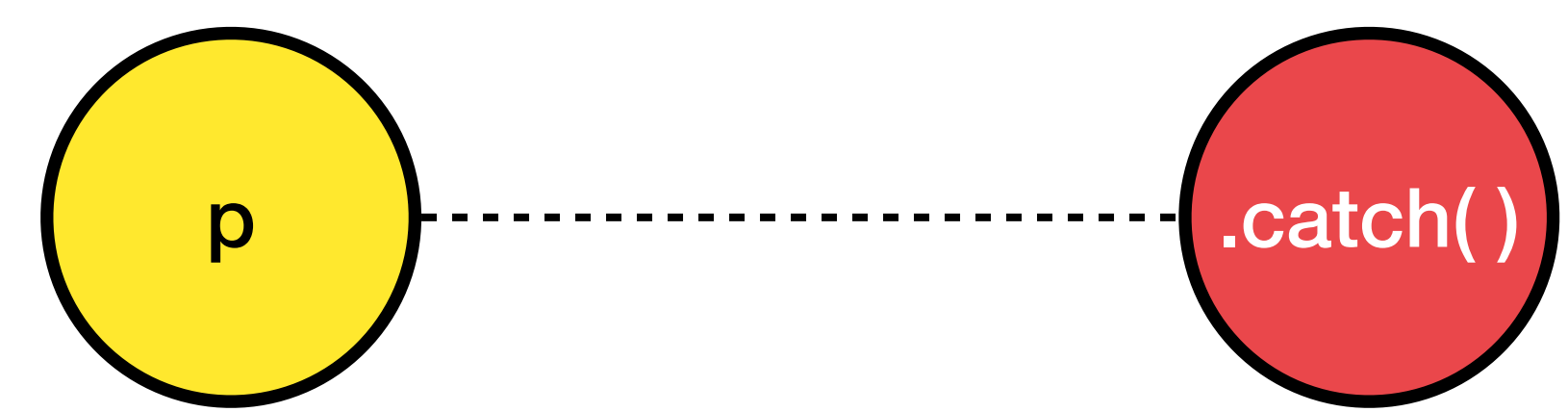
Um catch para todos controlar

```
const p = new Promise((resolve, reject) => {  
  return (Math.random() > 0.5) ? resolve('yay') : reject('no')  
})  
p  
.then(function acao1 (res) { console.log(`${res} da ação 1`); return res; })  
.then(function acao2 (res) { console.log(`${res} da ação 2`); return res; })  
.then(function acao3 (res) { console.log(`${res} da ação 3`); return res; })  
.catch(function erro (rej) { console.error(rej) })
```

Math.random > 0.5



Math.random <= 0.5

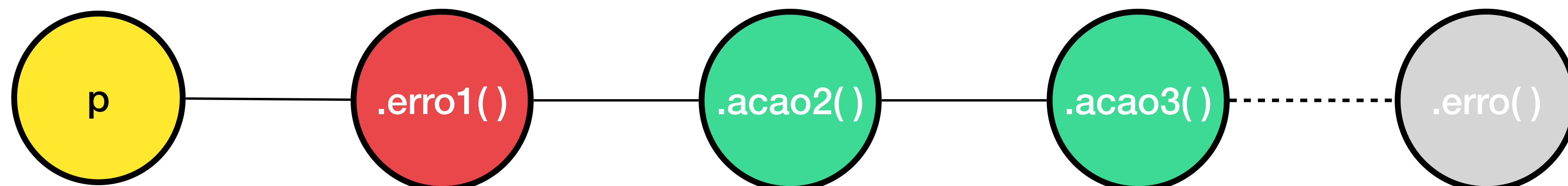


Encadeamento

Um catch para todos controlar

```
const p = new Promise((resolve, reject) => {
  return (Math.random() > 0.5) ? resolve('yay') : reject('no')
})
p
.then(function acao1 (res) { console.log(`${res} da ação 1`); return res; })
.catch(function erro1 (err) { console.error('Primeiro catch'); return 'Error'; })
.then(function acao2 (res) { console.log(`${res} da ação 2`); return res; })
.then(function acao3 (res) { console.log(`${res} da ação 3`); return res; })
.catch(function erro2 (rej) { console.error(rej) })
```

Math.random <= 0.5

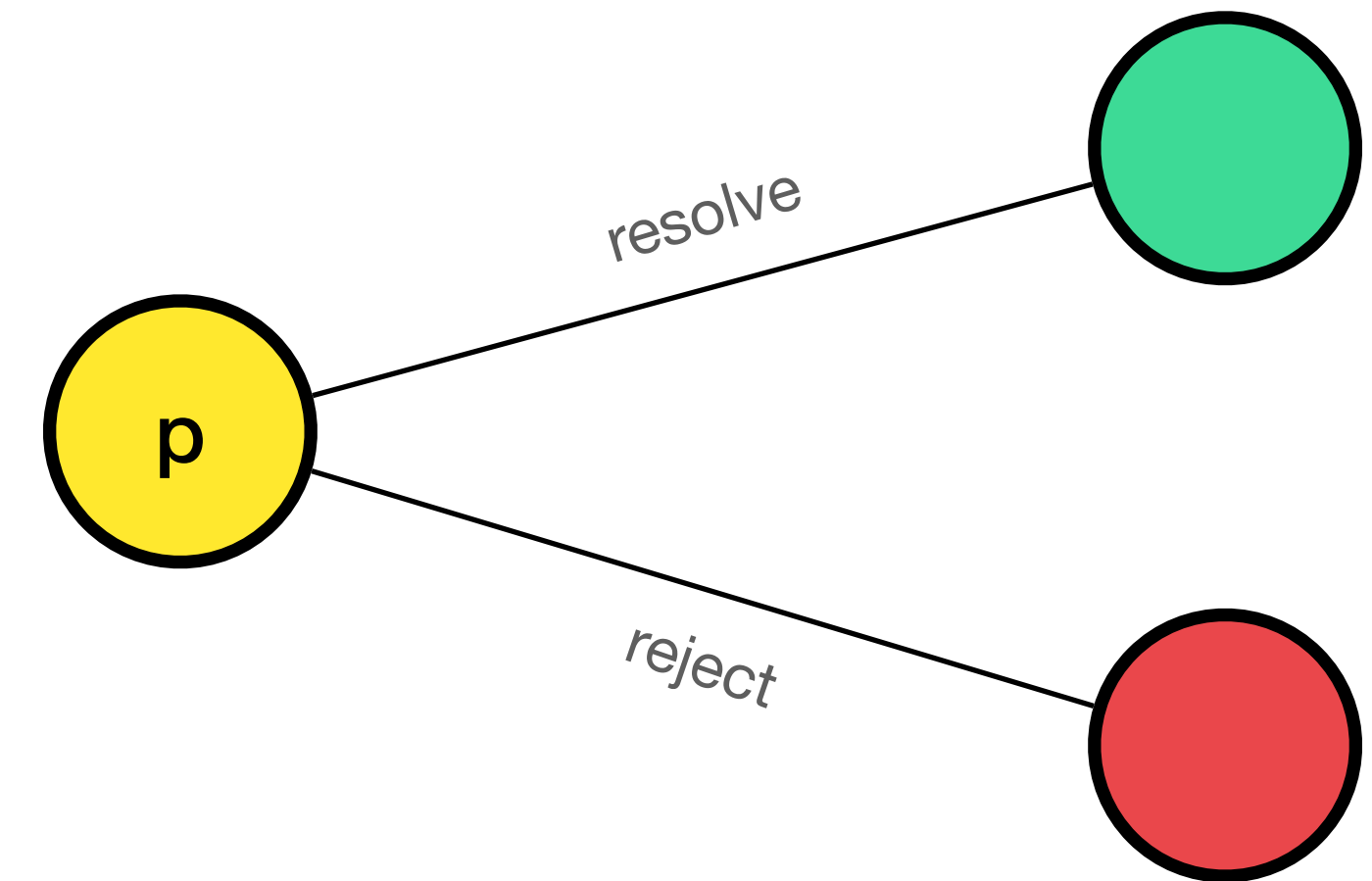


Encadeamento

Resumo

- Independente do tratamento que damos a Promise, ele sempre vai buscar o primeiro tratamento de erros disponível
 - Cada catch irá capturar o erro relativo às Promises anteriores
 - Em seguida, o valor que ele retornar será passado para a próxima Promise que executará normalmente.
- O catch não é universal, quando encadeados em outros then, o primeiro erro que acontece consome o primeiro catch e assim por diante
- Tal comportamento é difícil de se replicar com callbacks, visto que cada callback só iria capturar os erros de sua execução

Async/Await



Async/Await

Introdução

```
verificarCartao(cartao)
  .then((cartaoValido) => verificarSaldo(cartaoValido))
  .then((saldoAtual) => verificarFraude(cartaoValido, saldoAtual))
  .then((transacaoValida) => pagar(valor, cartao))
  .then((finalResult) => {
    console.log(`Pagamento realizado com sucesso`);
  })
  .catch(failureCallback);
```

```
function pagarComCartao(cartao, valor) {
  try {
    const cartaoValido = verificarCartao(cartao)
    const saldoAtual = verificarSaldo(cartaoValido)
    const transacaoValida = verificarFraude(cartaoValido, valor)
    const resultado = pagar(cartaoValido, transacaoValida)
    return `${resultado} realizada com sucesso"
  } catch (err) {
    failureCallback(err);
  }
}
```

Async/Await

Introdução

- A simetria entre os códigos anteriores culminaram na criação do *syntactic sugar*, `async/await`
 - Simplificam a escrita e leitura de código assíncrono

```
async function pagarComCartao(cartao, valor) {
  try {
    const cartaoValido = await verificarCartao(cartao)
    const saldoAtual = await verificarSaldo(cartaoValido)
    const transacaoValida = await verificarFraude(cartaoValido, valor)
    const resultado = await pagar(cartaoValido, transacaoValida)
    return `${resultado} realizada com sucesso`
  } catch (err) {
    failureCallback(err);
  }
}
```

Async/Await

Funções async

- A palavra *async* antes de uma função significa uma única coisa:
 - A função sempre retorna uma *promise*
 - Assim que o corpo da função retorna algo, essa promessa é resolvida
 - Se lançar uma exceção, a promessa é rejeitada

```
async function f() {  
  return "Oi mundo";  
}  
  
f().then(console.log); // Oi mundo
```

Async/Await

Funções async

- Uma função assíncrona pode esperar (*await*) uma ou mais *promises*
 - Apenas funções assíncronas tem essa capacidade

```
async function pagarComCartao(cartao, valor) {
  try {
    const cartaoValido = await verificarCartao(cartao)
    const saldoAtual = await verificarSaldo(cartaoValido)
    const transacaoValida = await verificarFraude(cartaoValido, valor)
    const resultado = await pagar(cartaoValido, transacaoValida)
    return `${resultado} realizada com sucesso`
  } catch (err) {
    failureCallback(err);
  }
}
```

Async/Await

Await

- A palavra-chave *await* faz com que o interpretador JS espere até que a promise alcance o estado *settled*
 - A função é suspensa até a resolução da *promise*
 - Não gera mais processamento
 - Nesse meio tempo outras tarefas são executadas
 - Após atingir o estado *settled*, o fluxo da função continua normalmente

Async/Await

Tratamento de erro

- Os possíveis erros são capturados por meio do bloco catch
- Async/await são formas mais elegantes de escrever código que lidam com *promises*

```
async function pagarComCartao(cartao, valor) {
  try {
    const cartaoValido = await verificarCartao(cartao)
    const saldoAtual = await verificarSaldo(cartaoValido)
    const transacaoValida = await verificarFraude(cartaoValido, valor)
    const resultado = await pagar(cartaoValido, transacaoValida)
    return `${resultado} realizada com sucesso`
  } catch (err) {
    failureCallback(err);
  }
}
```

Referências

- [Promises, async/await](#)
- [JavaScript Promises: an introduction](#)
- [Entendendo Promises de uma vez por todas](#)
- [Using promises](#)

Por hoje é só