Event Loop – Call back

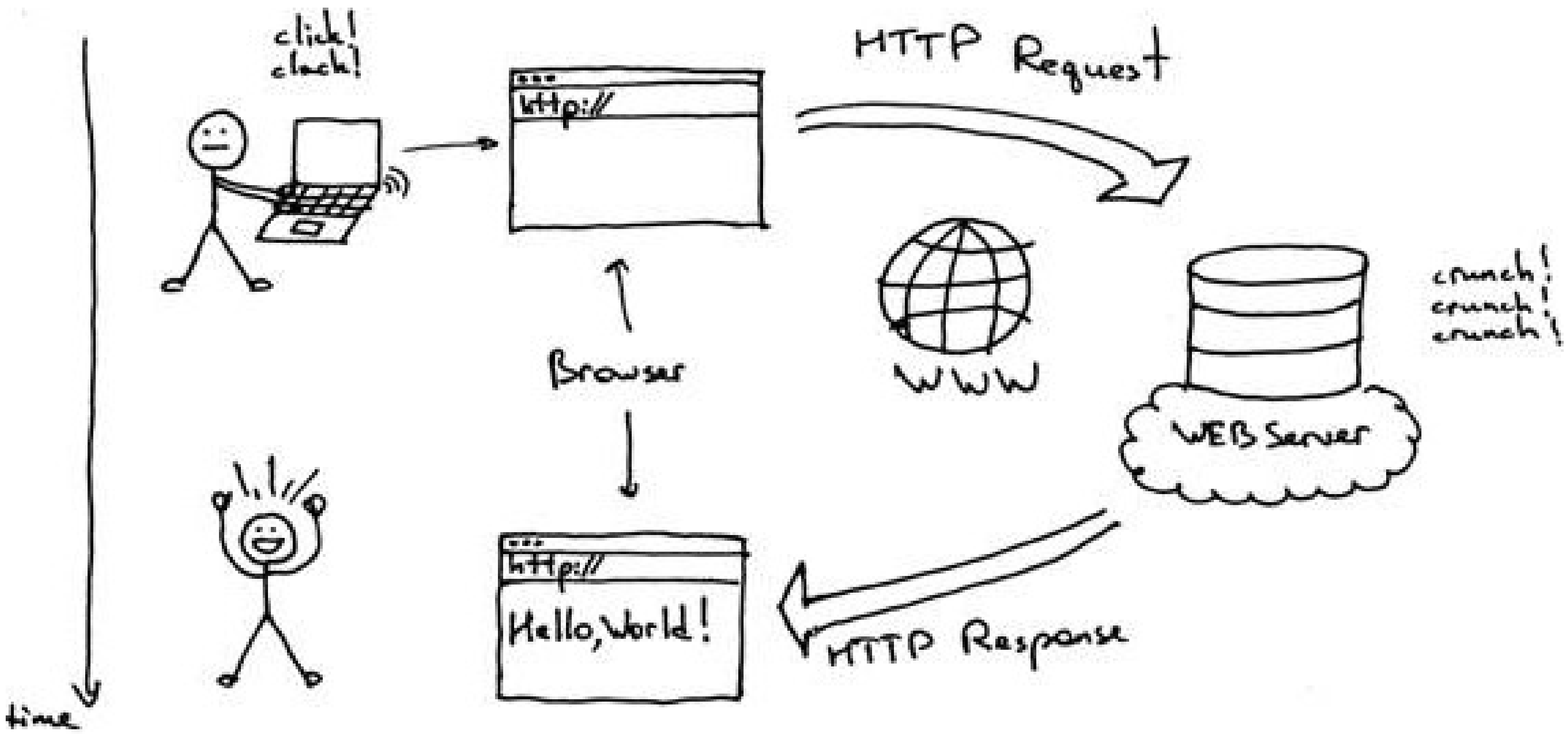
2f4869676820706572666f726d616e63652e2044656c6976657265642e2f4869676820706572666f726d616e63652e2044656c6976657265642e2f4869676820706572666f726d616e63652e2044656c6976657265642e2f4869676820706572666f726d616e63652e2044656c6976657265642e2f4869676820706572666f726d616e63652e2044656c6976657265642e2f48696768207

Conclusão:

- **Callbacks** nada mais são do que funções que levam algum tempo para produzir um resultado;
- Normalmente, esses retornos de chamada assíncronos são usados para acessar valores de bancos de dados, fazer download de imagens, ler arquivos etc.
-
- Como isso leva tempo para terminar, não podemos prosseguir para a próxima linha, pois isso pode gerar um erro, nem podemos pausar nosso programa;
- Portanto, precisamos armazenar o resultado e retornar a chamada quando estiver concluído;

Web Server

2f4869676820706572666f726d616e63652e2044656c6976657265642e2f4869676820706572666f726d616e63652e2044656c6976657265642e2f4869676820706572666f726d616e63652e2044656c6976657265642e2f4869676820706572666f726d616e63652e2044656c6976657265642e2f4869676820706572666f726d616e63652e2044656c6976657265642e2f48696768207

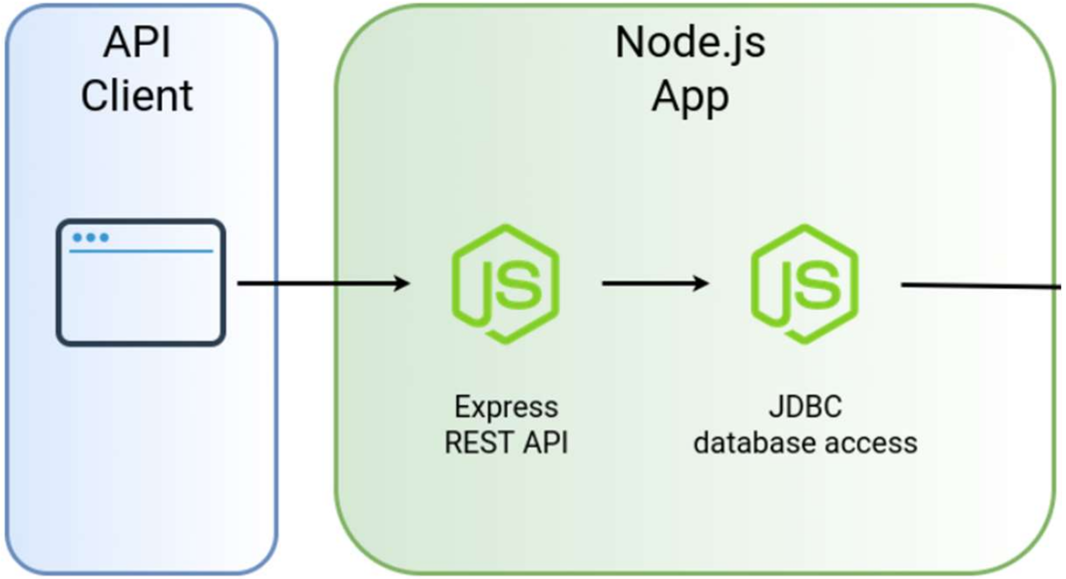


2f486967682070657266f726d616e63652e2044656c6976657265642e2f486967682070657266f726d616e63652e2044656c6976657265642e2f486967682070657266f726d616e63652e2044656c6976657265642e2f486967682070657266f726d616e63652e2044656c6976657265642e2f486967682070657266f726d616e63652e2044656c6976657265642e2f48696768207

Web Server

```
var http = require("http");

http.createServer(function (request, response) {
    // Send the HTTP header
    // HTTP Status: 200 : OK
    // Content Type: text/plain
    response.writeHead(200, {'Content-Type': 'text/plain'});

    // Send the response body as "Hello World"
    response.end('Hello World\n');
}).listen(8081);

// Console will print the message
console.log('Server running at http://127.0.0.1:8081/');
```

EXPRESS




```javascript
const express = require('express');
const app = express();

app.get('/ping', (request, response) => {
    response.send('pong');
});

app.listen(8080, 'localhost');
```

NPM

Gerenciador de Pacotes do NODE.JS

- O NPM (Node Package Manager) é o gerenciador de pacotes do Node.js;
- É o maior repositório de softwares do mundo;
- O pacote mais conhecido se chama **Express.js** e é um framework completo para desenvolvimento de aplicações Web;

npm –v
npm install <modulo>

EXPRESS

2f4869676820706572666f726d616e63652e2044656c6976657265642e2f4869676820706572666f726d616e63652e2044656c6976657265642e2f4869676820706572666f726d616e63652e2044656c6976657265642e2f4869676820706572666f726d616e63652e2044656c6976657265642e2f4869676820706572666f726d616e63652e2044656c6976657265642e2f4869676820706572666f726d616e63652e2044656c6976657265642e2f48696768207

Métodos de Resposta.

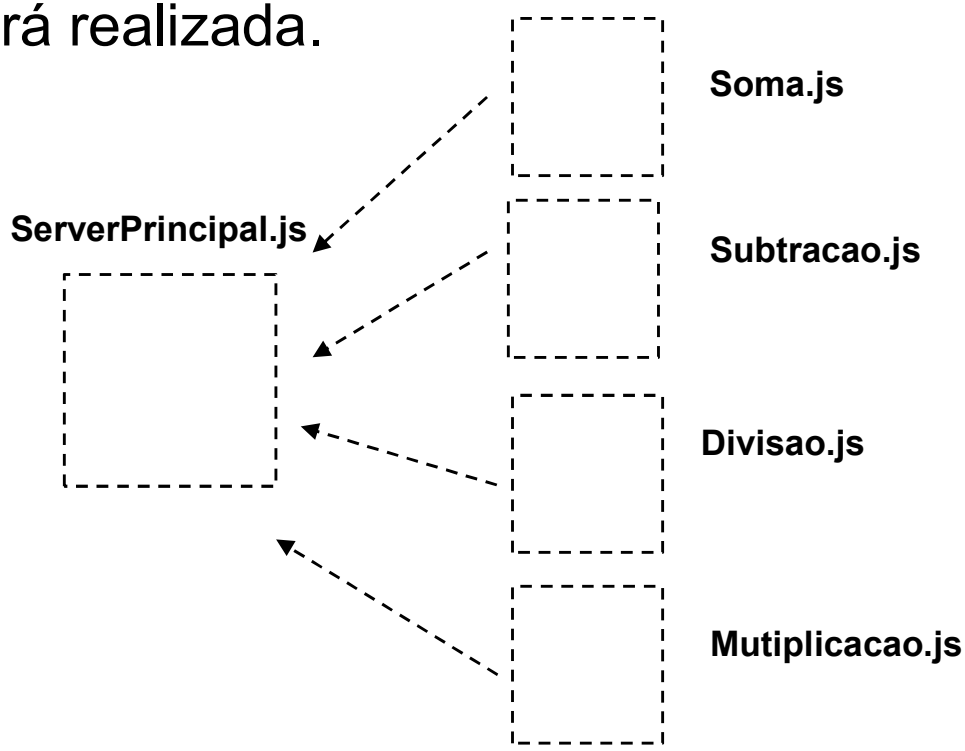
Os métodos de resposta (res) podem enviar uma resposta ao cliente, e finalizar o ciclo solicitação-resposta.

https://expressjs.com/pt-br/4x/api.html

Método	Descrição
res.download()	Solicita que seja efetuado o download de um arquivo
res.end()	Termina o processo de resposta.
res.json()	Envia uma resposta JSON.
res.jsonp()	Envia uma resposta JSON com suporta ao JSONP.
res.redirect()	Redireciona uma solicitação.
res.render()	Renderiza um modelo de visualização.
res.send()	Envia uma resposta de vários tipos.
res.sendFile	Envia um arquivo como um fluxo de octeto.
res.sendStatus()	Configura o código do status de resposta e envia a sua representação em sequência de caracteres como o corpo de resposta.

Atividade

- Crie uma API calculadora.
- A API deverá ter:
  - Pelo menos 4 end-points do tipo GET.
  - Usar os arquivos externos com as 4 operações básicas.
  - O Front-End deve enviar para o servidor Node os números e o tipo de operação que será realizada.



MySQL


```
var mysql = require('mysql');

var con = mysql.createConnection({
  host: "localhost",
  user: "root",
  password: "root",
  database: "mydb"
});

con.connect(function(err) {
  if (err) throw err;
  console.log("Connected!");
});
```

MySQL

2f4869676820706572666f726d616e63652e2044656c6976657265642e2f4869676820706572666f726d616e63652e2044656c6976657265642e2f4869676820706572666f726d616e63652e2044656c6976657265642e2f4869676820706572666f726d616e63652e2044656c6976657265642e2f4869676820706572666f726d616e63652e2044656c6976657265642e2f48696768207

```
var mysql = require('mysql');

var con = mysql.createConnection({
  host: "localhost",
  user: "root",
  password: "root",
  database: "mydb"
});

con.connect(function(err) {
  if (err) throw err;
  //Select all customers and return the result object:
  con.query("SELECT * FROM aluno", function (err, result, fields) {
    if (err) throw err;
    console.log(result);
  });
});
```

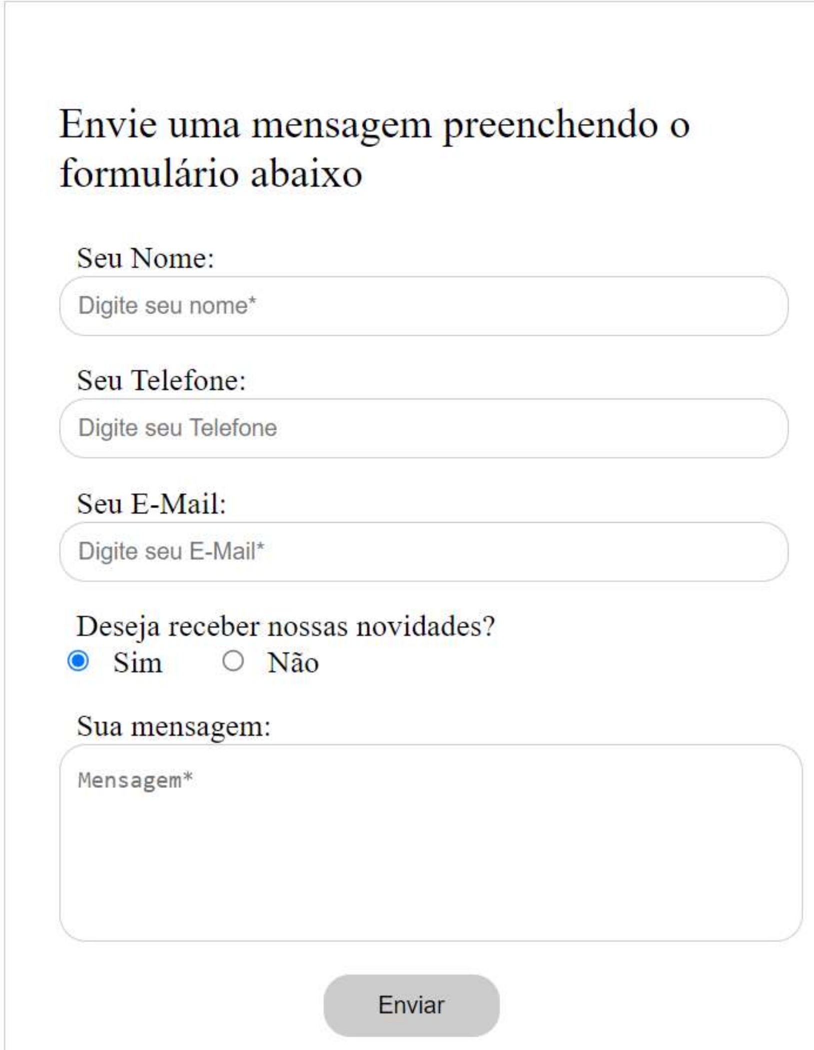
MySQL


```
con.connect(function(err) {
  if (err) throw err;
  console.log("Connected!");
  //Make SQL statement:
  var sql = "INSERT INTO aluno
     (idAluno, nome, tipo_aluno, tipoBolsa, valorMensalidade) VALUES ?";
  //Make an array of values:
  var values = [
    ['101','John',' ','B50',500],
    ['102','Peter',' ','B90',100],
    ['103','Chuck',' ',' ',1000],
    ['104','Viola',' ','B10',900]
  ];
  //Execute the SQL statement, with the value array:
  con.query(sql, [values], function (err, result) {
    if (err) throw err;
    console.log("Number of records inserted: " + result.affectedRows);
  });
});
```

Atividade

Criar uma tabela usuários no mysql.
Criar a API que vai receber os dados enviados do formulário.
Salvar os dados na tabela.

PassagemDeValores.html
Node_Sparametro2.js



Axios

- **O que é o Axios?**

  - Axios é um cliente HTTP baseado em promise para o browser e Node. js.
  - O Axios facilita o envio de solicitações HTTP assíncronas para endpoints REST e a execução de operações CRUD.

```
axios.get(...)
    .then(...)
    .catch(error => {
        // what now?
    })
```

  - Node_server3.js   node_Axios3.js

Axios

2f486967682070657266f726d616e63652e2044656c6976657265642e2f486967682070657266f726d616e63652e2044656c6976657265642e2f486967682070657266f726d616e63652e2044656c6976657265642e2f486967682070657266f726d616e63652e2044656c6976657265642e2f486967682070657266f726d616e63652e2044656c6976657265642e2f486967682070657266f726d616e63652e2044656c6976657265642e2f48696768207

```
axios.post('/user_login', {
        username: ‘peterParker',
        password: 'spiderman’
    })
.then(function (response) {
        console.log(response);
    })
.catch(function (error) {
        console.log(error);
    });
```

Axios


```
axios.get(“/list”)
.then(function (response) {
        const lista = response[0].data
        console.log(lista);
      })
.catch(function (error) {
        console.log(error);
      });
```

Atividade

- Crie uma API **axios** para consumir a API VIACEP.
  - Criar uma pagina web simples para informar o CEP.
  - Criar a tabela de CEP no mysql.
  - Fazer GET na viacep.
  - Recuperar os dados.
  - Conectar no mySQL.
  - Salvar em uma tabela o cep recuperado.

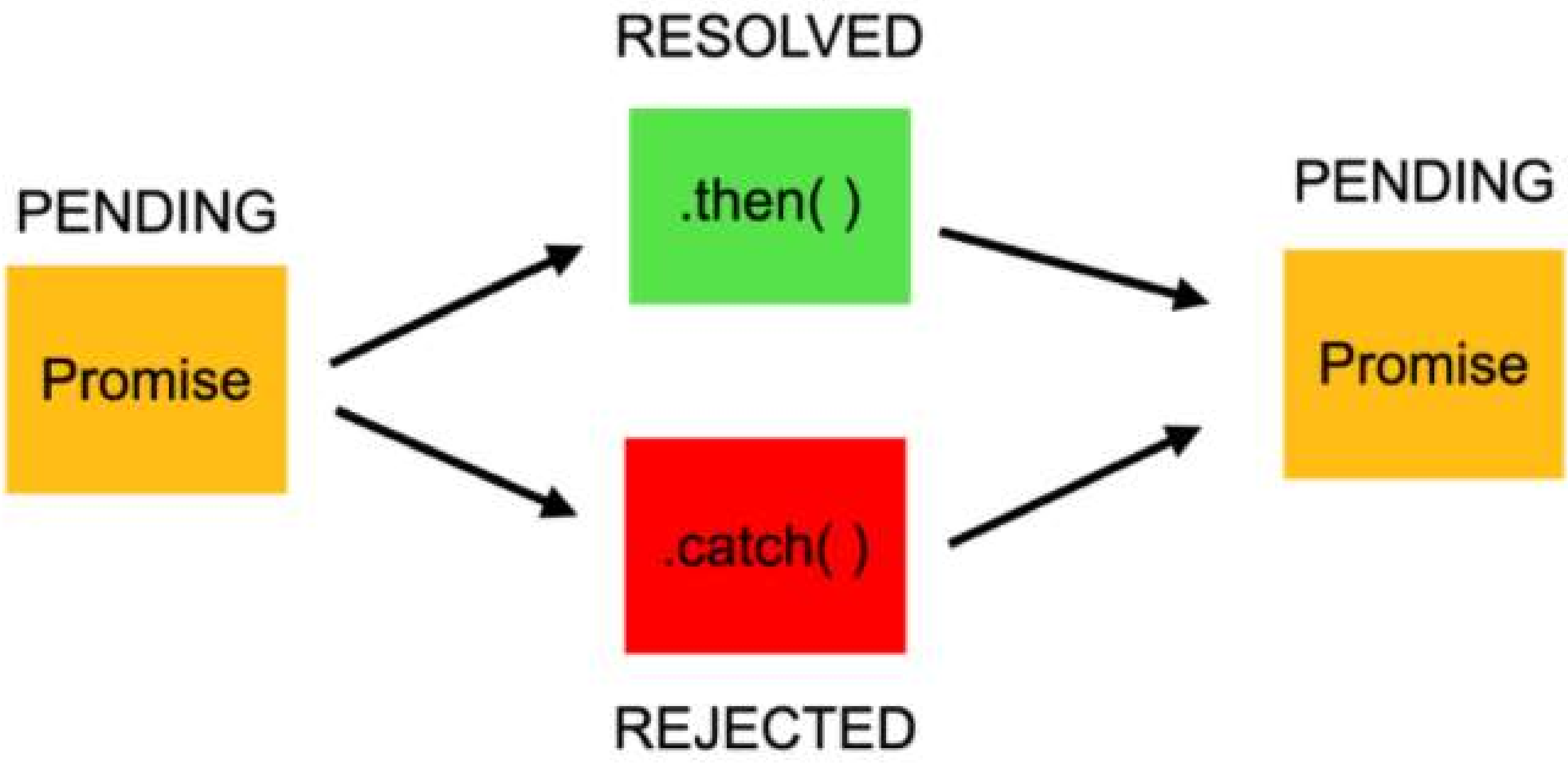
- https://viacep.com.br/ws/01001000/json/

Event Loop – Call back

Promise:

- JavaScript fornece uma maneira fácil de escapar de um inferno de callback. Isso é feito por **promises**.
- Uma **promise** é um objeto retornado de qualquer função assíncrona, ao qual métodos de retorno de chamada podem ser adicionados com base no resultado da função anterior.
- As **promises** usam o método **.then()** para tratar retornos de chamada assíncronos. Podemos encadear quantas chamadas de retorno quisermos e a ordem também é estritamente mantida.
- As **promises** usam o método **.fetch()** para buscar um objeto da rede. Ele também usa o método **.catch()** para capturar qualquer exceção quando qualquer bloco falhar.
- Portanto, essas **promises** são colocadas na fila de eventos para que não bloqueiem o código JS subsequente. Além disso, quando os resultados são retornados, a fila de eventos termina suas operações.
- Existem também outras palavras-chave e métodos úteis, como **async**, **wait**, **settimeout()** para simplificar e fazer melhor uso dos retornos de chamada.

Promises



Promises

2f4869676820706572666f726d616e63652e2044656c6976657265642e2f4869676820706572666f726d616e63652e2044656c6976657265642e2f4869676820706572666f726d616e63652e2044656c6976657265642e2f4869676820706572666f726d616e63652e2044656c6976657265642e2f48696768207

- **Promises** vem do termo “promessa”, que representa um valor que pode estar disponível em algum momento no futuro, no presente, ou nunca estar disponível.

- Ele é um objeto utilizado em processamento **assíncrono**.

- Um **promises** representa um proxy para um valor não necessariamente conhecido, permitindo uma associação de métodos de tratamento para eventos em uma ação assíncrona na hipótese de sucesso ou falha, permitindo que o método **assíncrono** retorne uma “promessa” ao valor em algum momento no futuro.

- Elas não eram nativas do **JavaScript** até o **ES6**, quando houve uma implementação oficial na linguagem, antes delas, a maioria das funções usavam ***callbacks***.

Promises

2f486967682070657266f726d616e63652e2044656c6976657265642e2f486967682070657266f726d616e63652e2044656c6976657265642e2f486967682070657266f726d616e63652e2044656c6976657265642e2f486967682070657266f726d616e63652e2044656c6976657265642e2f486967682070657266f726d616e63652e2044656c6976657265642e2f48696768207

- **Estados de uma Promise**

  - *Pending:* O estado inicial da Promise, ela foi iniciada mas ainda não foi realizada nem rejeitada

  - *Fulfilled:* Sucesso da operação, é o que chamamos de uma Promise **realizada** (ou, em inglês, *resolved)* — eu, pessoalmente, prefiro o termo **resolvida**.

  - *Rejected:* Falha da operação, é o que chamamos de uma Promise **rejeitada** (em inglês, *rejected*)

Promises

2f4869676820706572666f726d616e63652e2044656c6976657265642e2f4869676820706572666f726d616e63652e2044656c6976657265642e2f4869676820706572666f726d616e63652e2044656c6976657265642e2f4869676820706572666f726d616e63652e2044656c6976657265642e2f4869676820706572666f726d616e63652e2044656c6976657265642e2f4869676820706572666f726d616e63652e2044656c6976657265642e2f4869676820706572666f726d616e63652e2044656c6976657265642e2f48696768207

- NODE_PromisessoPar.js
- NODE_PromisessoPares.js
- NODE_PromisessoParesTeste

```
soAceitaPares(2)
    .then(result => console.log("Promise resolved: " + result))
    .catch(error => console.log("Promises rejected: " + error));

console.log("teste");
```

```
//index.js
function soAceitaPares(numero){
    const promise = new Promise( (resolve, reject) => {
                    if (numero % 2 === 0) {
                        const resultado = 'Viva, é par!';
                        resolve(resultado);
                    }
                    else {
                        reject(new Error("Você passou um número impar!"));
                    }
                });
    return promise;
}
```

Promises

2f486967682070657266f726d616e63652e2044656c6976657265642e2f486967682070657266f726d616e63652e2044656c6976657265642e2f486967682070657266f726d616e63652e2044656c6976657265642e2f486967682070657266f726d616e63652e2044656c6976657265642e2f486967682070657266f726d616e63652e2044656c6976657265642e2f48696768207

- **NODE_Promisedivide.js**
- **NODE_PromisedivideTeste.js**

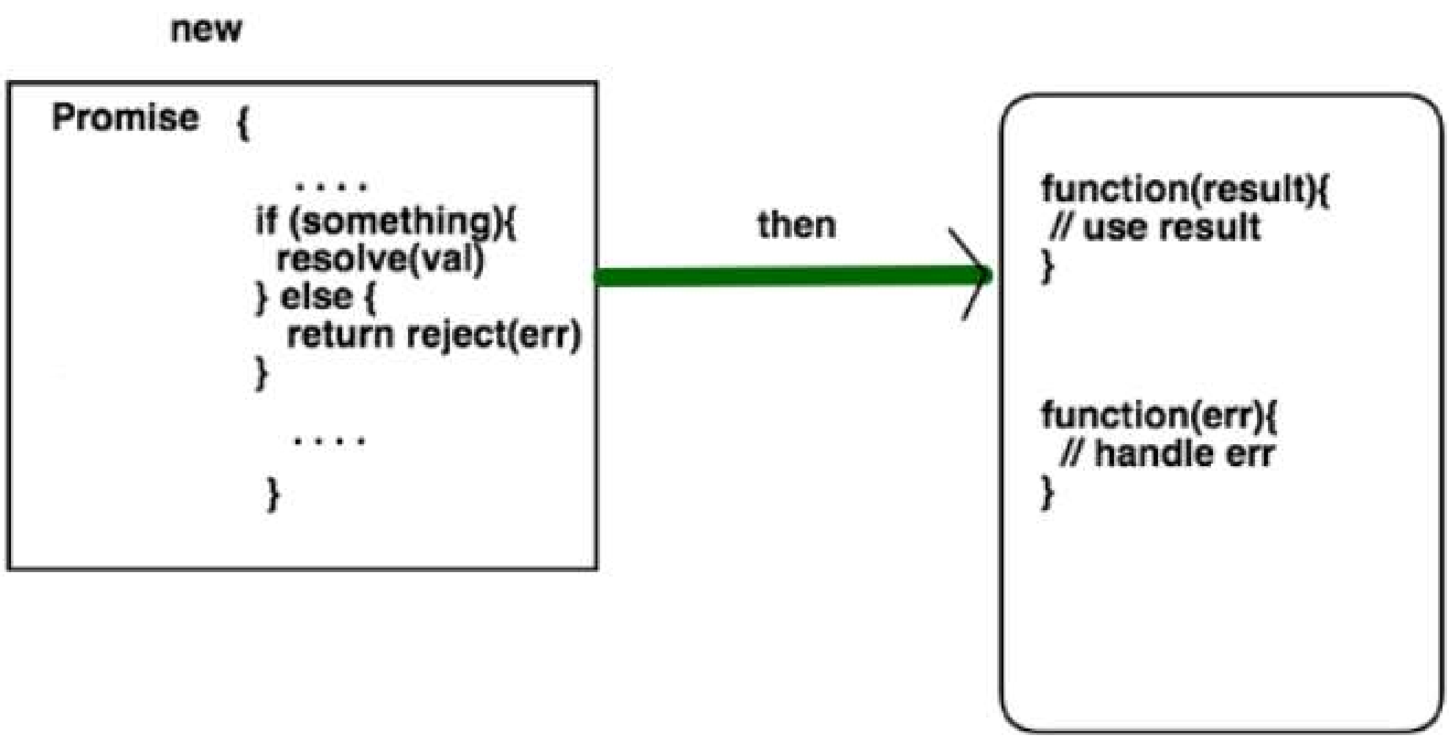
```
function dividePelaMetade(numero){
    if(numero % 2 !== 0)
        return Promise.reject(new Error("Não posso dividir um número impar!"));
    else
        return Promise.resolve(numero / 2);
}
```

```
if(soAceitaPares(numero))
   dividePelaMetade(numero);
```

```
const numero = 2;
soAceitaPares(2)
    .then(result => dividePelaMetade(numero))
    .then(result2 => console.log("A metade de " + numero + " é " + result2))
    .catch(error => console.log("Promises rejected: " + error));

console.log("teste");
```

Promises

2f4869676820706572666f726d616e63652e2044656c6976657265642e2f4869676820706572666f726d616e63652e2044656c6976657265642e2f4869676820706572666f726d616e63652e2044656c6976657265642e2f4869676820706572666f726d616e63652e2044656c6976657265642e2f4869676820706572666f726d616e63652e2044656c6976657265642e2f48696768207



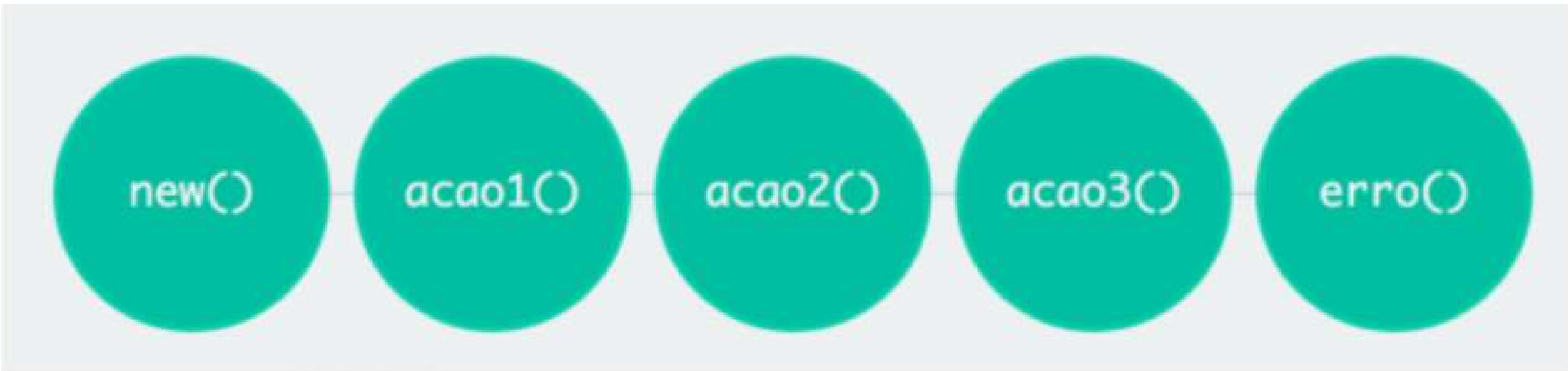
What happens if you try to access the value from promise before it is

Promises


```
// Criando uma promise
const p = new Promise((resolve, reject) => {
  try {
    resolve(funcaoX())
  } catch (e) {
    reject(e)
  }
})

// Executando uma promise
p
  .then((parametros) => /* sucesso */)
  .catch((erro) => /* erro */)

// Tratando erros e sucessos no then
p
  .then(resposta => { /* tratar resposta */ }, erro => { /* tratar erro */ })
```

Promises

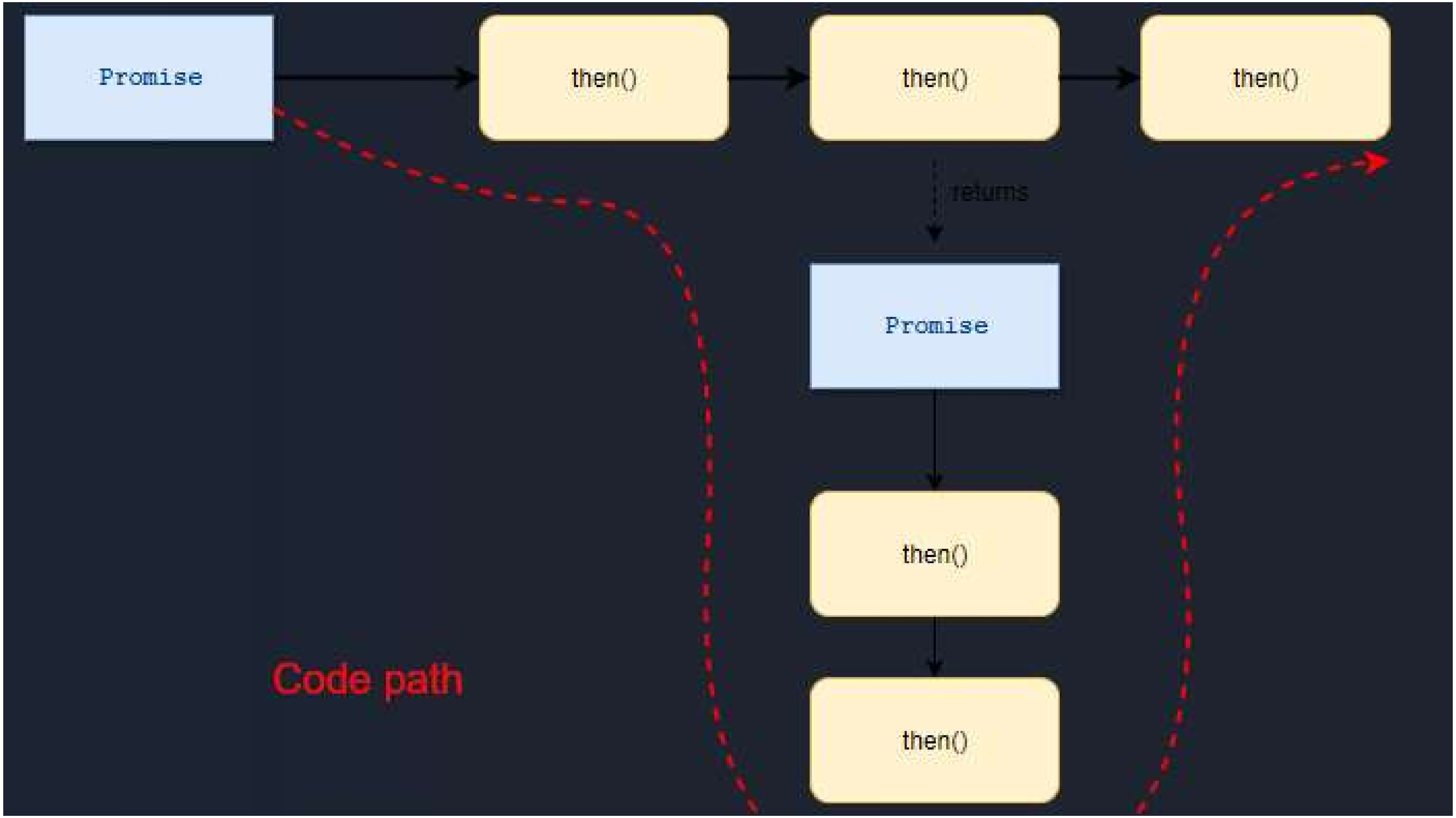

```
const p = new Promise((resolve, reject) => {
  if (Math.random() > 0.5) resolve('yay')
  reject('no')
})

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) })
```



NODE_promise.js, 5

Promises

2f486967682070657266f726d616e63652e2044656c6976657265642e2f486967682070657266f726d616e63652e2044656c6976657265642e2f486967682070657266f726d616e63652e2044656c6976657265642e2f486967682070657266f726d616e63652e2044656c6976657265642e2f48696768207



Promises


```javascript
const innerPromise = (val) => Promise.resolve(console.log(val + 1))
                              .then(() => console.log(val + 2))
                              .then(() => {
                                  console.log(5)
                                  return 5
                              })

const chain = Promise.resolve(console.log(0))
.then(() => console.log(1))
.then(() => console.log(2) || 2)
.then(innerPromise)
.then((five) => console.log(five + 1))

```

NODE_promisses3.js