MODELO DE EXECUÇÃO DO JAVASCRIPT

(AULA 22)

CURSO BÁSICO DE PROGRAMAÇÃO COM JAVASCRIPT

MAYARA MARQUES

mmrosatab@gmail.com

JS

SUMÁRIO

- Características do Javascript
- Modelo de execução do JavaScript
  - Call Stack
  - Task Queue
  - Microtask Queue
  - Event Loop
- Mão na massa

CARACTERÍSTICAS DO JAVASCRIPT

1. Single-threaded: executa uma operação por vez.

2. Non-Blocking: permite lidar com **tarefas demoradas sem bloquear a execução principal**. O código não trava esperando uma operação demorada terminar. O JavaScript delega a execução dessa operação demorada e continua rodando.

3. Concurrent: Várias operações assíncronas podem ser iniciadas e gerenciadas ao mesmo tempo, mas não são realmente executadas em paralelo.

CARACTERÍSTICAS DO JAVASCRIPT

```
const fetchPokemonData = async () => {
  try {
    const result = await
fetch('https://pokeapi.co/api/v2/pokemon/ditto' )
    const data = await result.json()
    console.log(data.abilities)

  } catch (error) {
    console.log(error)
  }
}
console.log('Instruction before fetch' )
fetchPokemonData ()
console.log('Instruction after fetch' )
```

Requisição HTTP

Saída:

```
Instruction before fetch
Instruction after fetch
[
  {
    ability: { name: 'limber', url:
'https://pokeapi.co/api/v2/ability/7/'  },
    is hidden: false,
    slot: 1
  },
  {
    ability: { name: 'imposter', url:
'https://pokeapi.co/api/v2/ability/150/'  },
    is hidden: true,
    slot: 3
  }
]
```

JavaScript **não bloqueia** a execução para aguardar a resposta dos dados do fetch

Se o JavaScript **não bloqueia a execução** para aguardar a resposta dos dados do fetch, como ele sabe retomar a execução da função quando resposta é recebida?

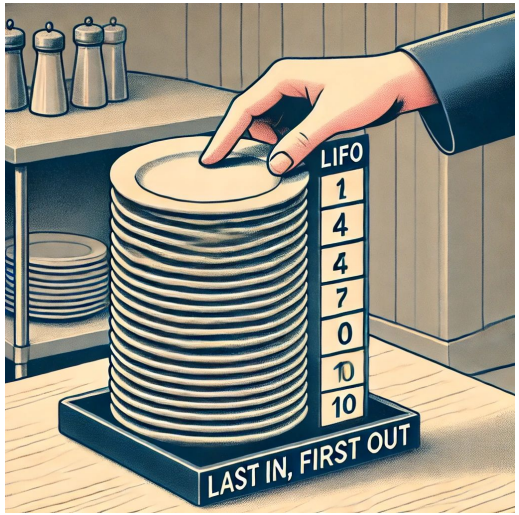
MODELO DE EXECUÇÃO DO JAVASCRIPT

O JavaScript permite trabalhar com tarefas assíncronas e síncronas por meio de gerenciamento através de uma pilha chamada **Call Stack**, duas filas chamadas **Task Queue** e **Microtask Queue e** mecanismos do ambiente de execução**,** chamados **Web Api's**.

MODELO DE EXECUÇÃO DO JAVASCRIPT

Conceito importante!

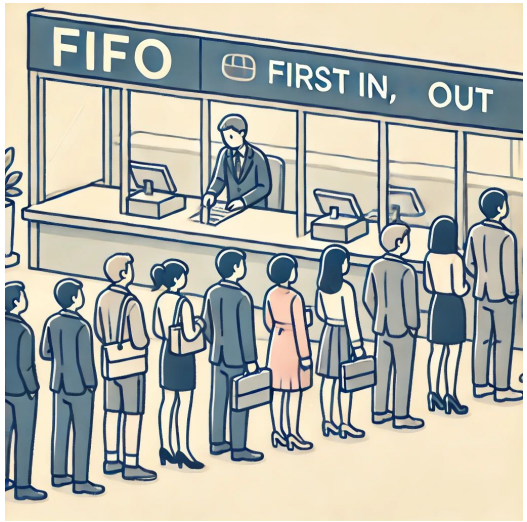
As **pilhas** seguem o princípio **LIFO (Last In, First Out)**, ou seja, **o último elemento inserido é o primeiro a ser removido**.



MODELO DE EXECUÇÃO DO JAVASCRIPT

Conceito importante!

As **filas** seguem o princípio **FIFO (First In, First Out)**, ou seja, **o primeiro elemento inserido é o primeiro a ser removido**.



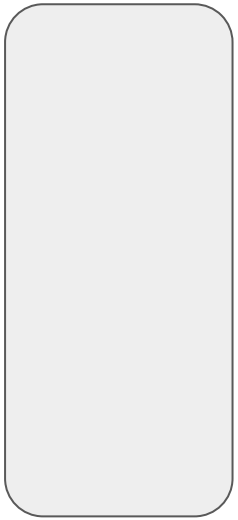
CALL STACK

Call Stack (Pilha de Chamadas)

O call stack é uma estrutura de dados do tipo pilha (LIFO - Last In, First Out) usada para **gerenciar a execução de funções no programa**.

Seu objeto armazenar as funções de modo a gerenciar/controlar a ordem de execução das funções.

Call Stack (LIFO)



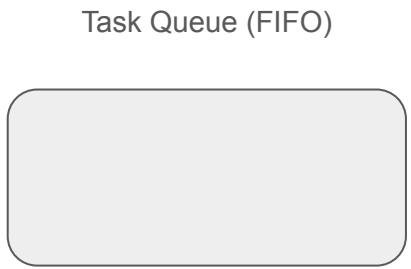
CALL STACK

Call Stack (Pilha de Chamadas)

1. Quando uma função é chamada, ela é empilhada no call stack.
2. Quando a função termina sua execução, ela é removida do topo do call stack.
3. A call stack **prioriza funções síncronas**.

Task Queue

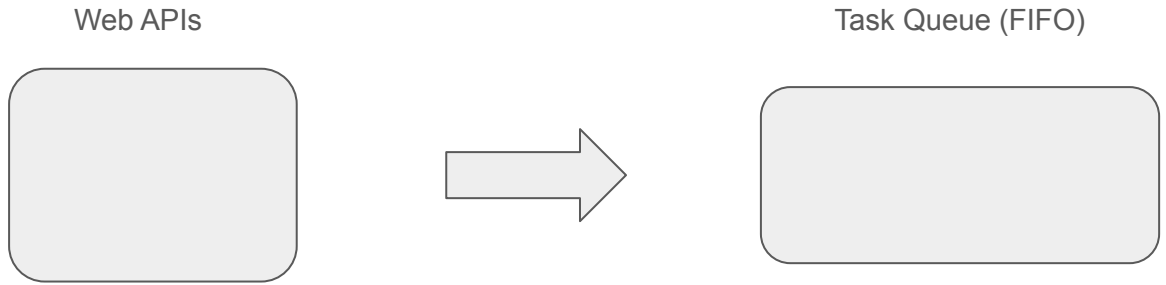
A task queue é fila (FIFO - First In, First Out) onde os **callbacks** das **tarefas assíncronas comuns**, como setTimeout, eventos do DOM e requisições de rede (fetch ou XMLHttpRequest), esperam para serem executadas.



WEB API'S

Para onde vão as tarefas assíncronas comuns enquanto ainda não obtiveram resposta?

As tarefas assíncronas comuns **ficam aguardando em mecanismos do ambiente de execução** até que sejam concluídas e, em seguida, enviadas para **task queue**. Esses mecanismos são conhecidos como **Web API's** (no navegador) ou APIs do ambiente (em Node.js).



WEB API'S

1. No navegador:

As tarefas assíncronas comuns, como setTimeout, eventos do DOM, e requisições de rede (fetch ou XMLHttpRequest), são gerenciadas por Web API's disponibilizadas pelo navegador.

- Exemplo de Web APIs no navegador:
  - Timer (setTimeout, setInterval).
  - APIs de rede (fetch, XMLHttpRequest).
  - Funções de manipulação de eventos do DOM.
  - APIs de animação (requestAnimationFrame).

WEB API'S

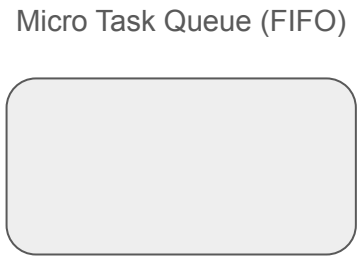
2. Em Node.js:

No caso de Node.js, as tarefas assíncronas comuns são gerenciadas pelo libuv, uma biblioteca que fornece um loop de eventos para operações como:

- I/O (entrada e saída de arquivos).
- Timers.
- Operações de rede.
- Assim como no navegador, as operações assíncronas são gerenciadas fora do call stack até que sejam concluídas.

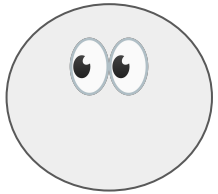
MICROTASK QUEUE

A microtask queue é uma **fila** (FIFO - First In, First Out) **de tarefas prioritárias**, usada para **Promises** e outras microtasks. A **microtask queue tem prioridade sobre a task queue**. Em outras palavras, se ambas as filas estiverem com tarefas, as tarefas da microtask queue serão executadas primeiro.



EVENT LOOP

Event Loop



Event Loop (Laço de Eventos)

O event loop é responsável por monitorar o call stack e as task queue e microtask queue para garantir que o JavaScript continue processando eventos e executando tarefas.

- Enquanto o call stack não estiver vazio, o event loop não fará nada.
- Quando o call stack estiver vazio, o event loop prioriza as tarefas da microtask queue para serem executadas primeiro e na sequência verifica a task queue.

RESUMO

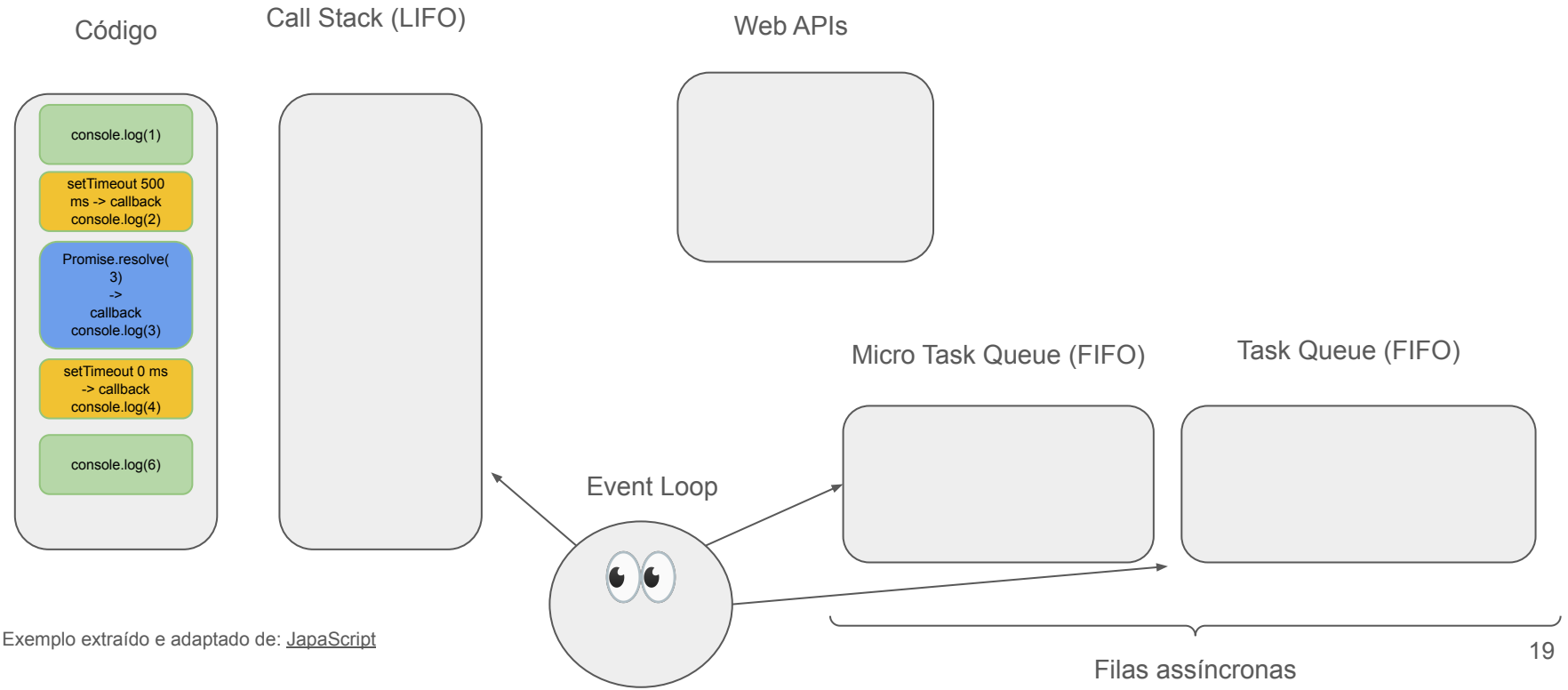
1. JavaScript executa código assíncrono **sem bloquear** o thread principal.
2. O código **síncrono sempre tem prioridade sobre o código assíncrono**.
3. Toda execução começa pela call stack.
    a. O JavaScript lê o código de cima para baixo.
    b. As operações síncronas são executadas imediatamente na call stack.
4. Somente após a call stack esvaziar, o event loop verifica as filas assíncronas.
    a. As microtasks (promises, mutations, etc.) são processadas primeiro.
    b. Depois, as tasks normais (setTimeout, I/O, eventos) são processadas.

MODELO DE EXECUÇÃO DO JAVASCRIPT

```
console.log(1)

setTimeout(() => {
  console.log(2)
}, 500)

Promise.resolve(3).then((result) => {
  console.log(result)
})

setTimeout(() => {
  console.log(4)
}, 0)

console.log(6)
```

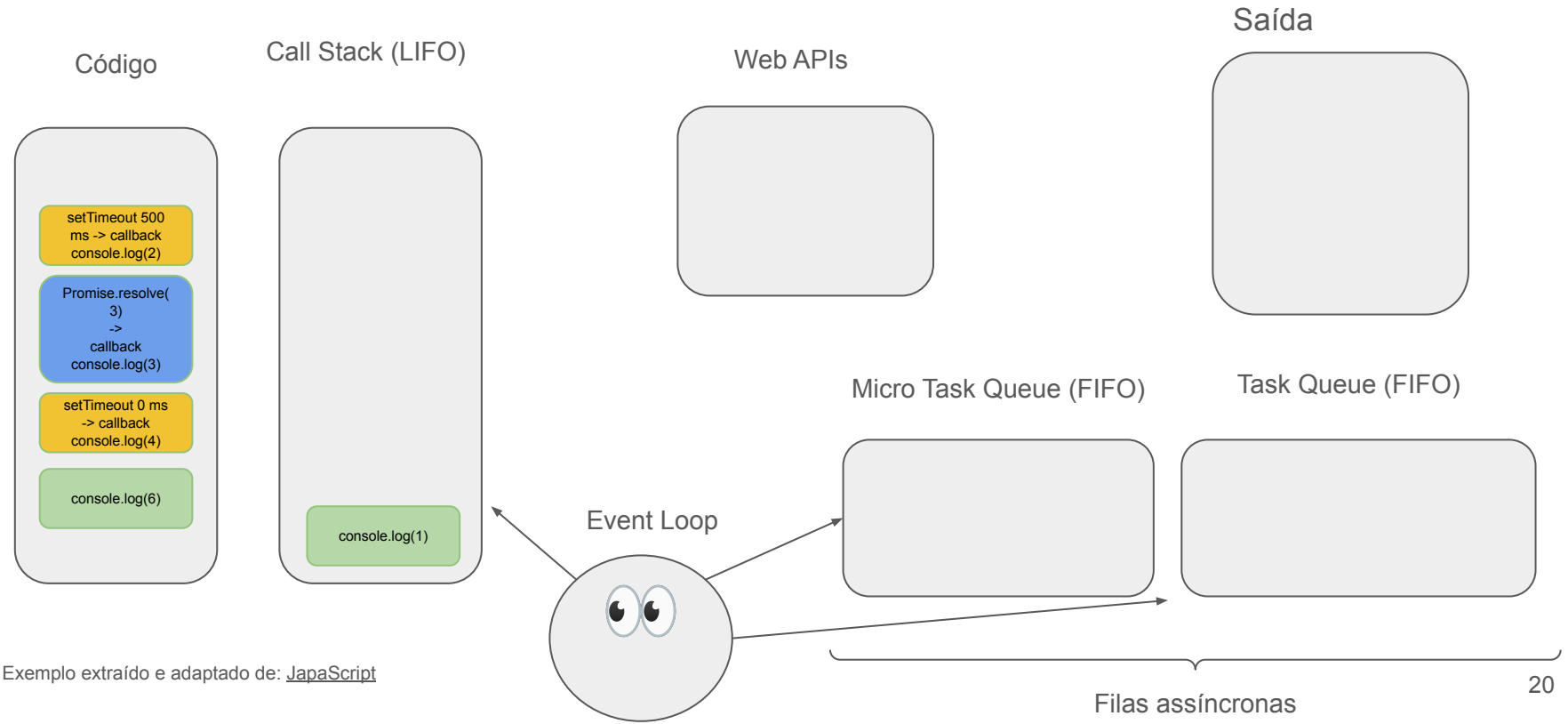
Código extraído de: JapaScript

MODELO DE EXECUÇÃO DO JAVASCRIPT



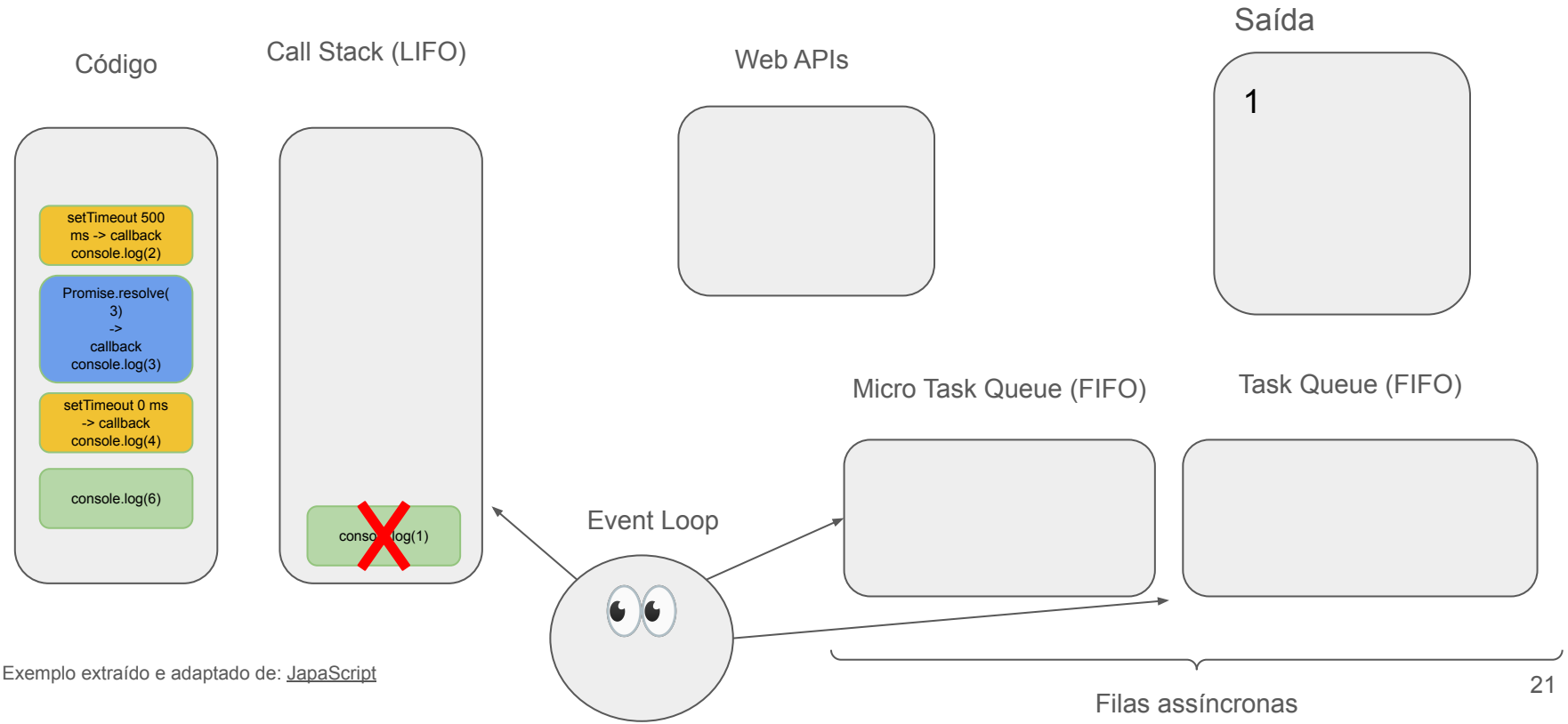
Exemplo extraído e adaptado de: JapaScript

MODELO DE EXECUÇÃO DO JAVASCRIPT



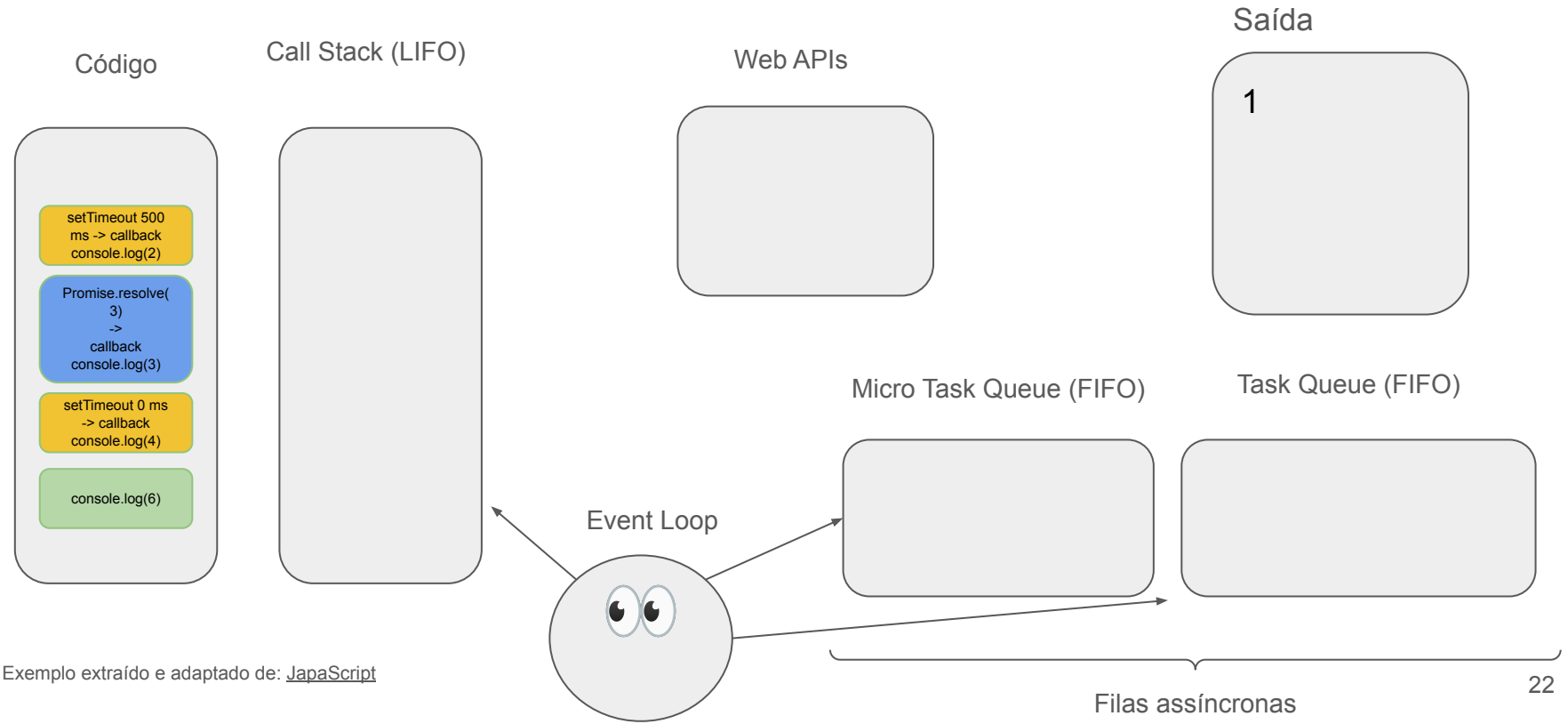
Exemplo extraído e adaptado de: JapaScript

MODELO DE EXECUÇÃO DO JAVASCRIPT



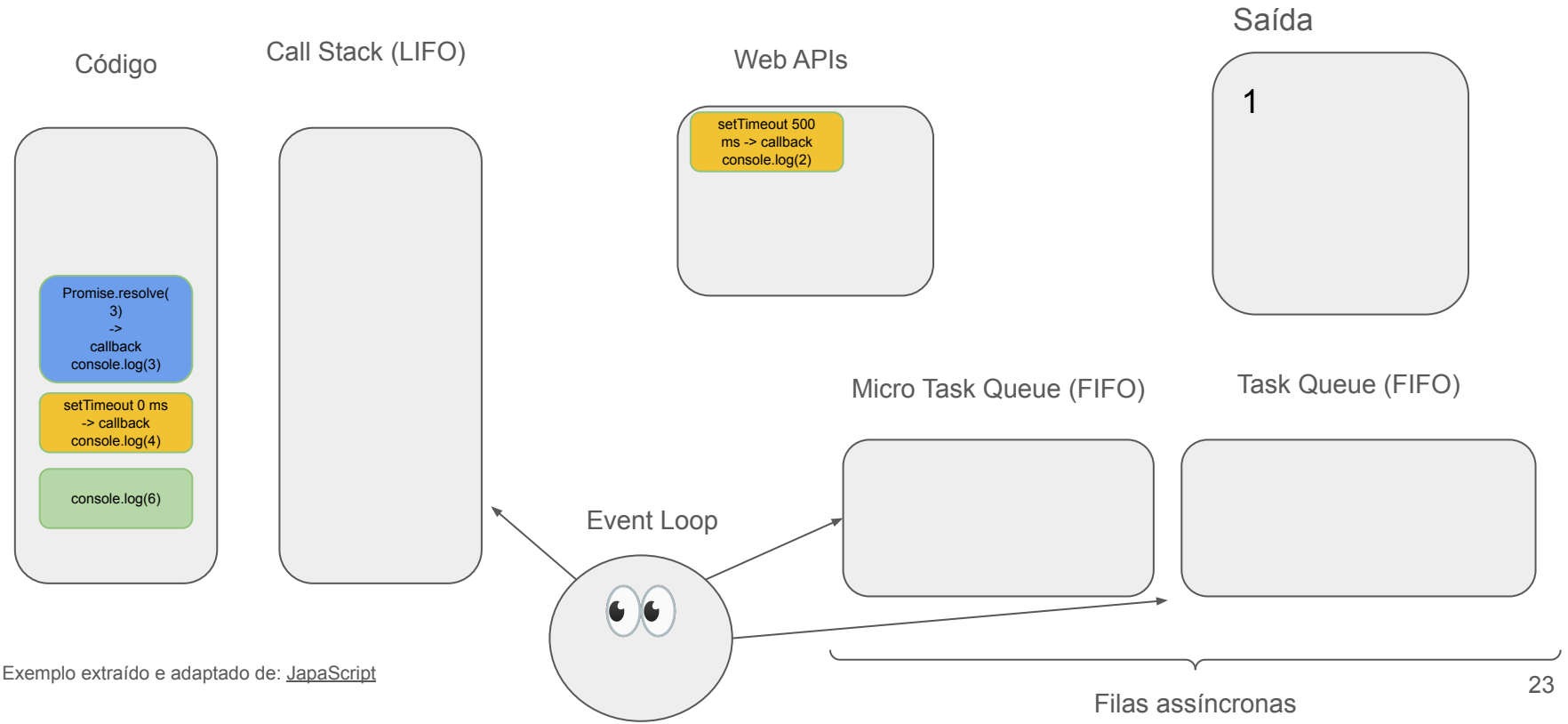
Exemplo extraído e adaptado de: JapaScript

MODELO DE EXECUÇÃO DO JAVASCRIPT



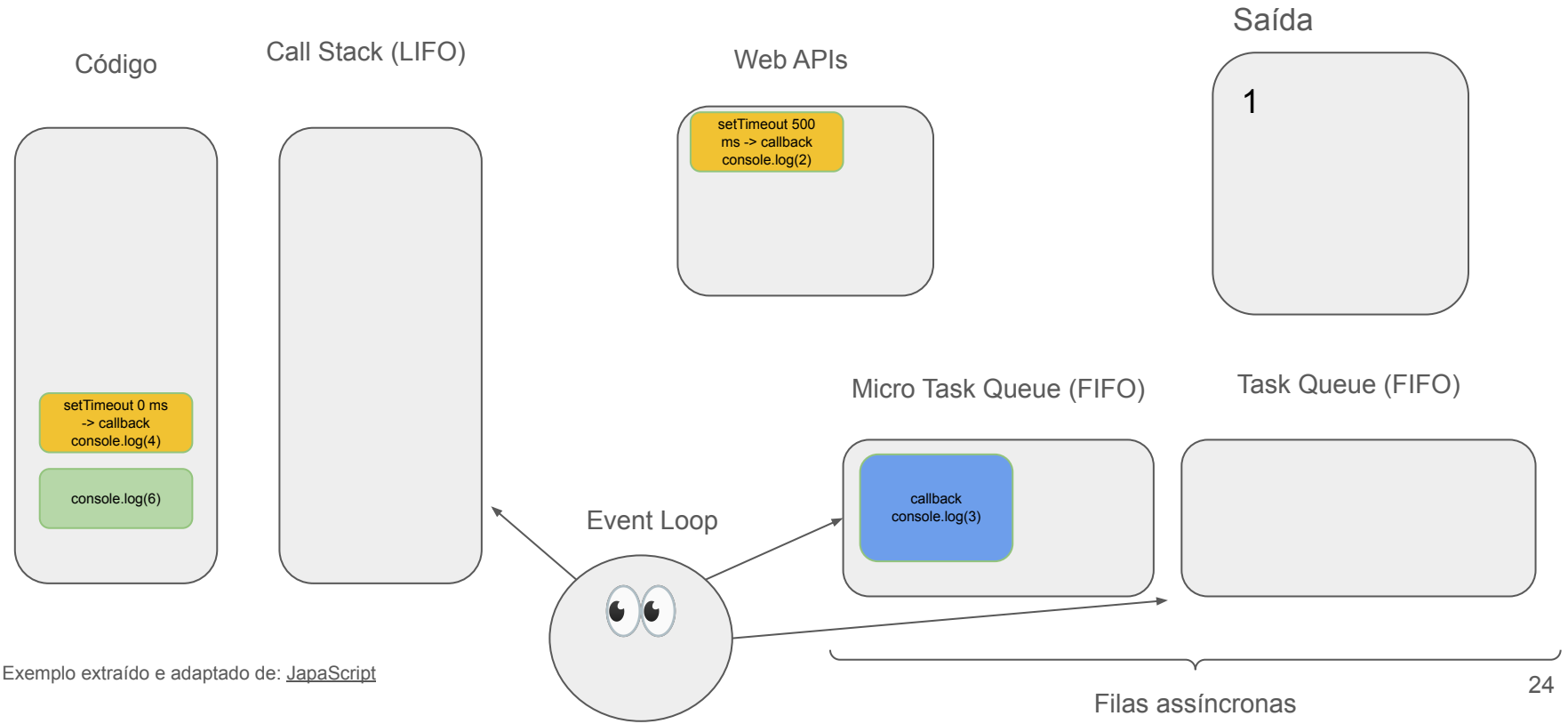
Exemplo extraído e adaptado de: JapaScript

MODELO DE EXECUÇÃO DO JAVASCRIPT



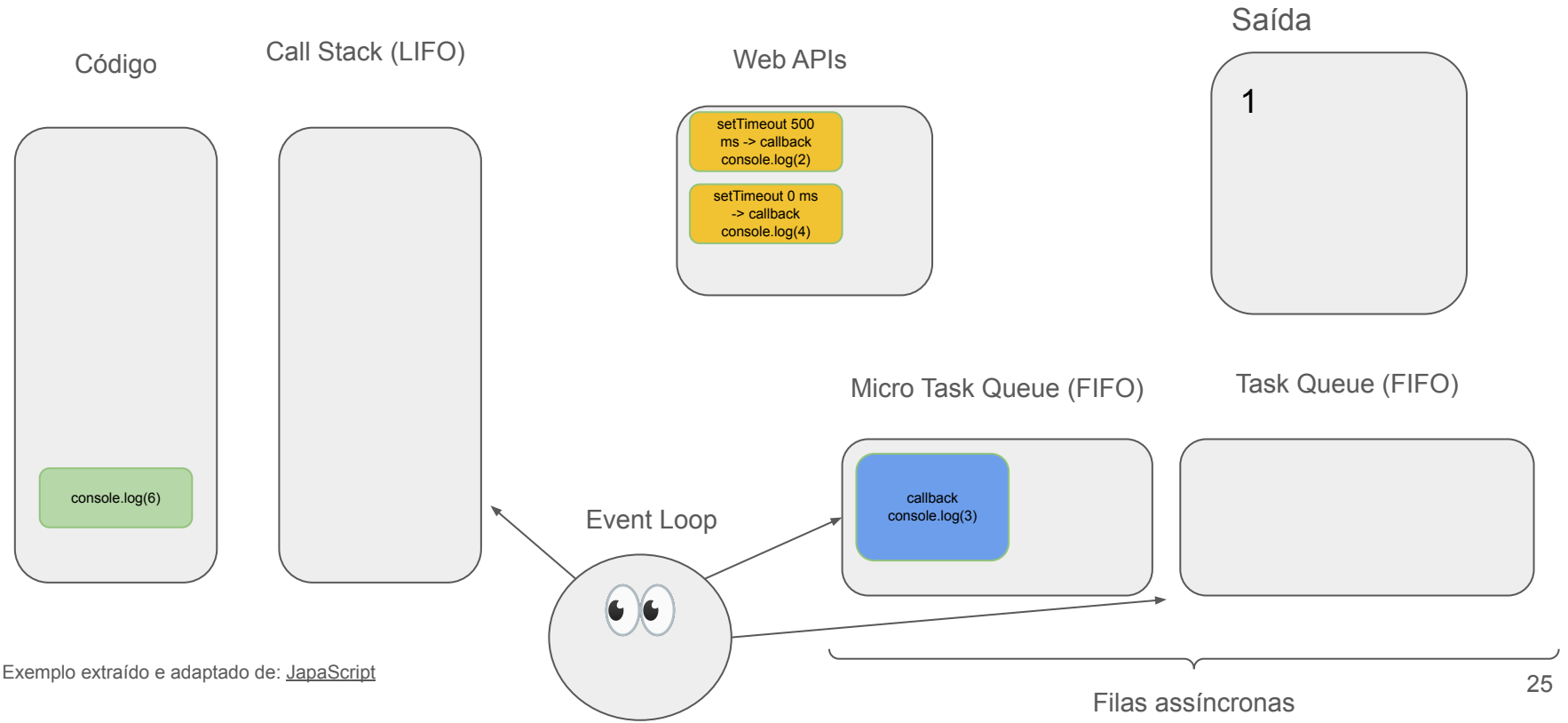
Exemplo extraído e adaptado de: JapaScript

MODELO DE EXECUÇÃO DO JAVASCRIPT



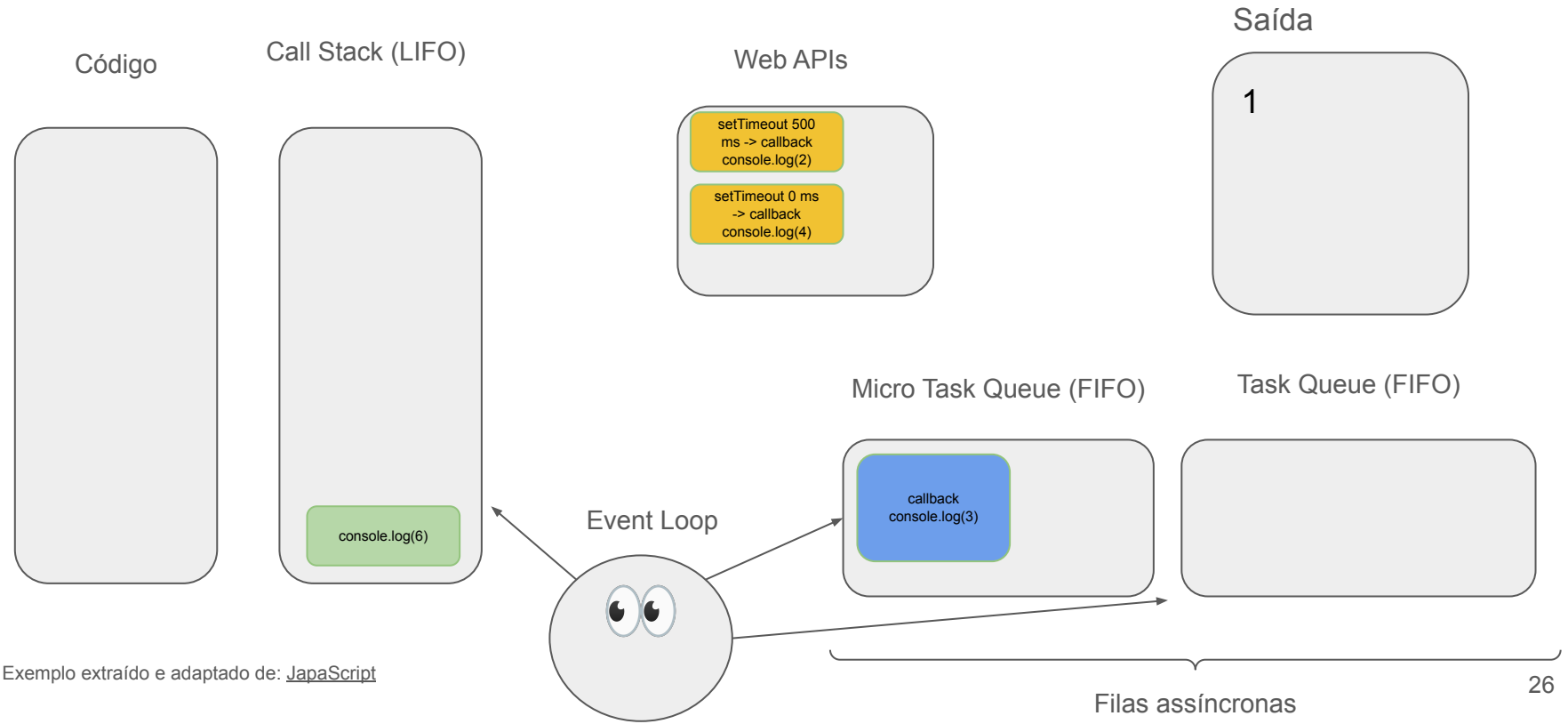
Exemplo extraído e adaptado de: JapaScript

MODELO DE EXECUÇÃO DO JAVASCRIPT



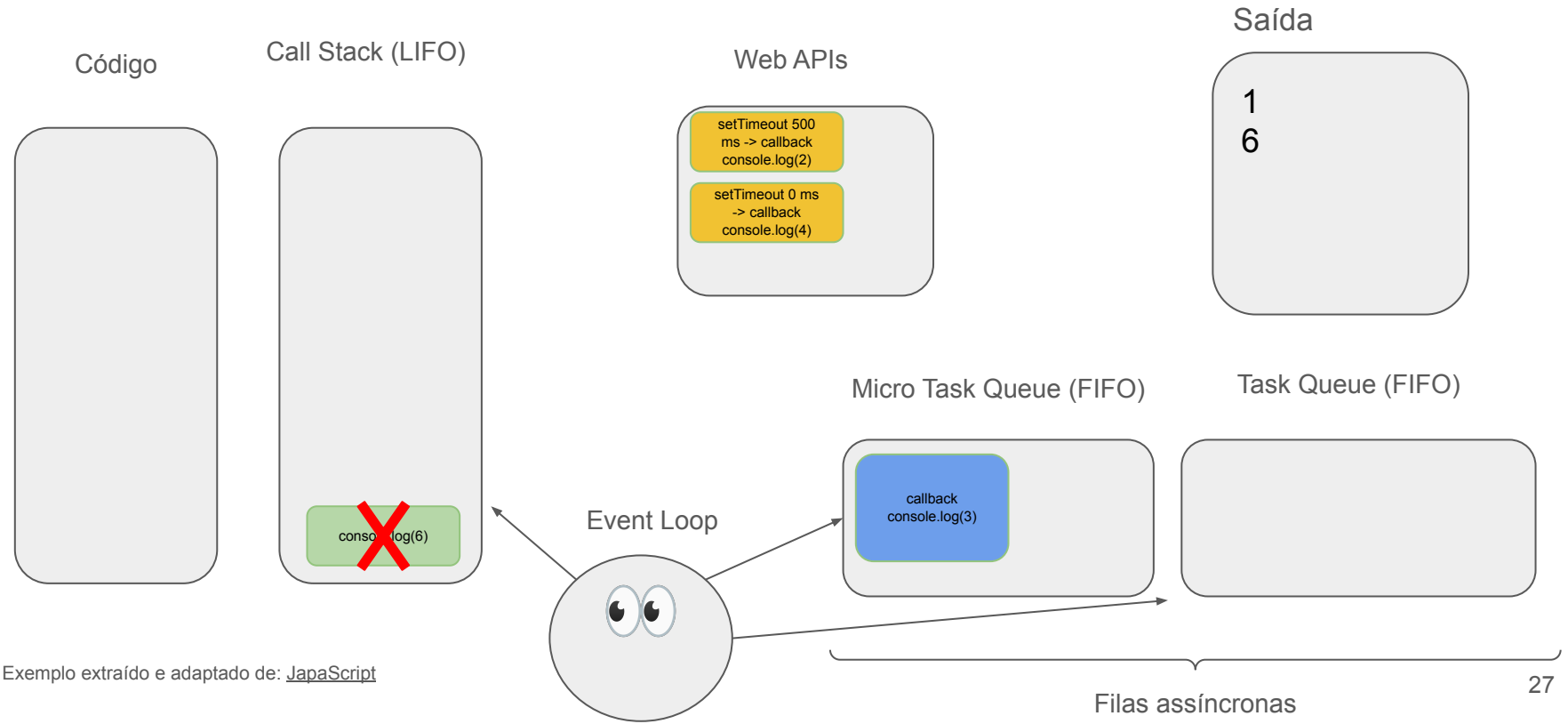
Exemplo extraído e adaptado de: JapaScript

MODELO DE EXECUÇÃO DO JAVASCRIPT



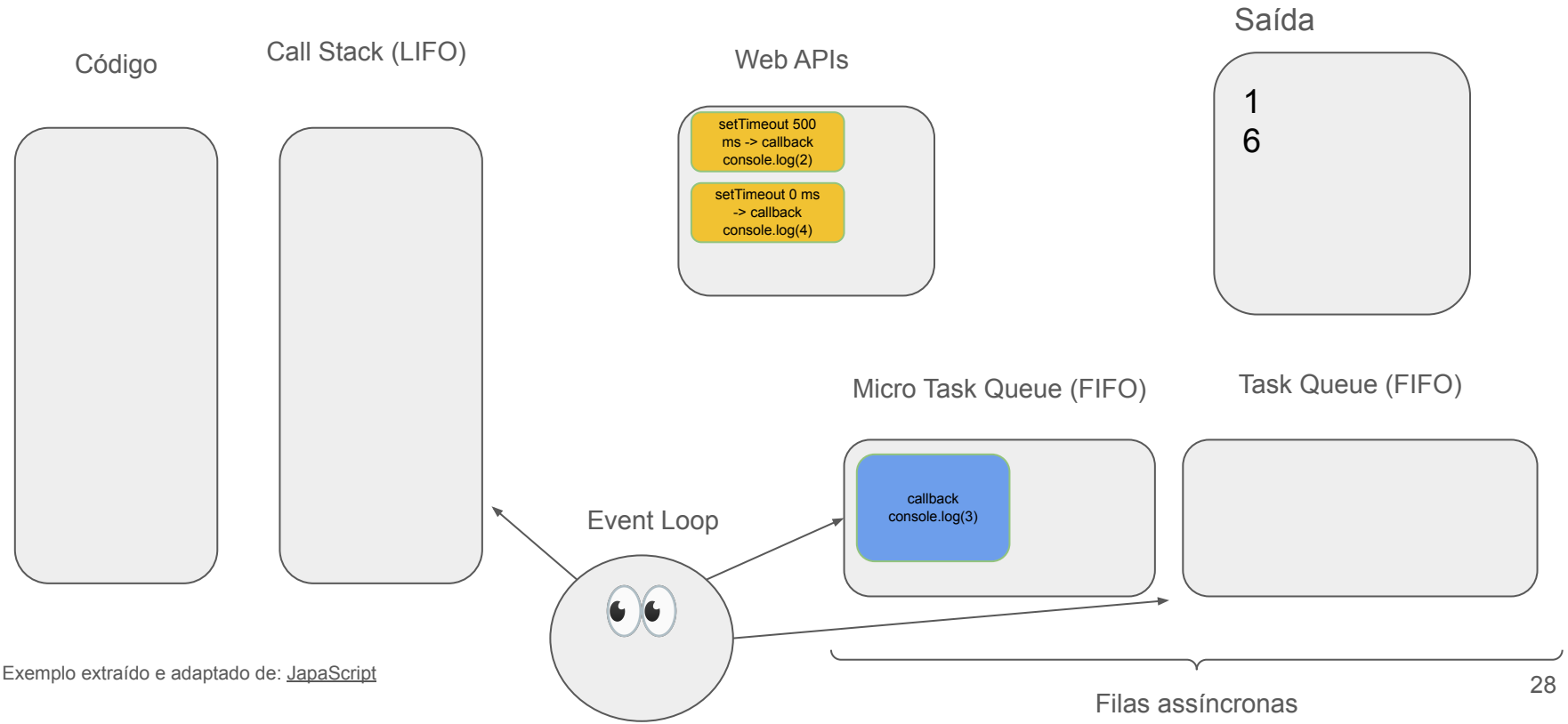
Exemplo extraído e adaptado de: JapaScript

MODELO DE EXECUÇÃO DO JAVASCRIPT



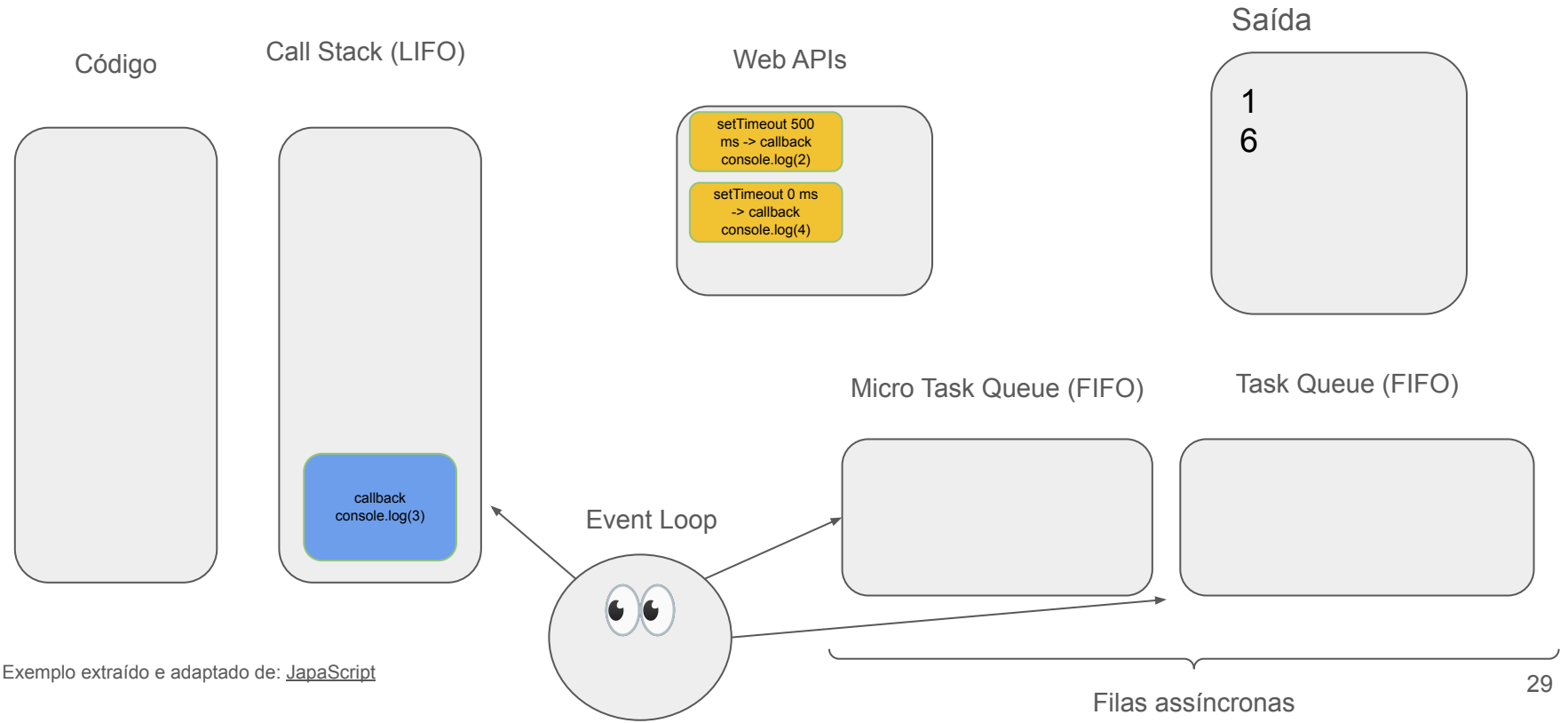
Exemplo extraído e adaptado de: JapaScript

MODELO DE EXECUÇÃO DO JAVASCRIPT



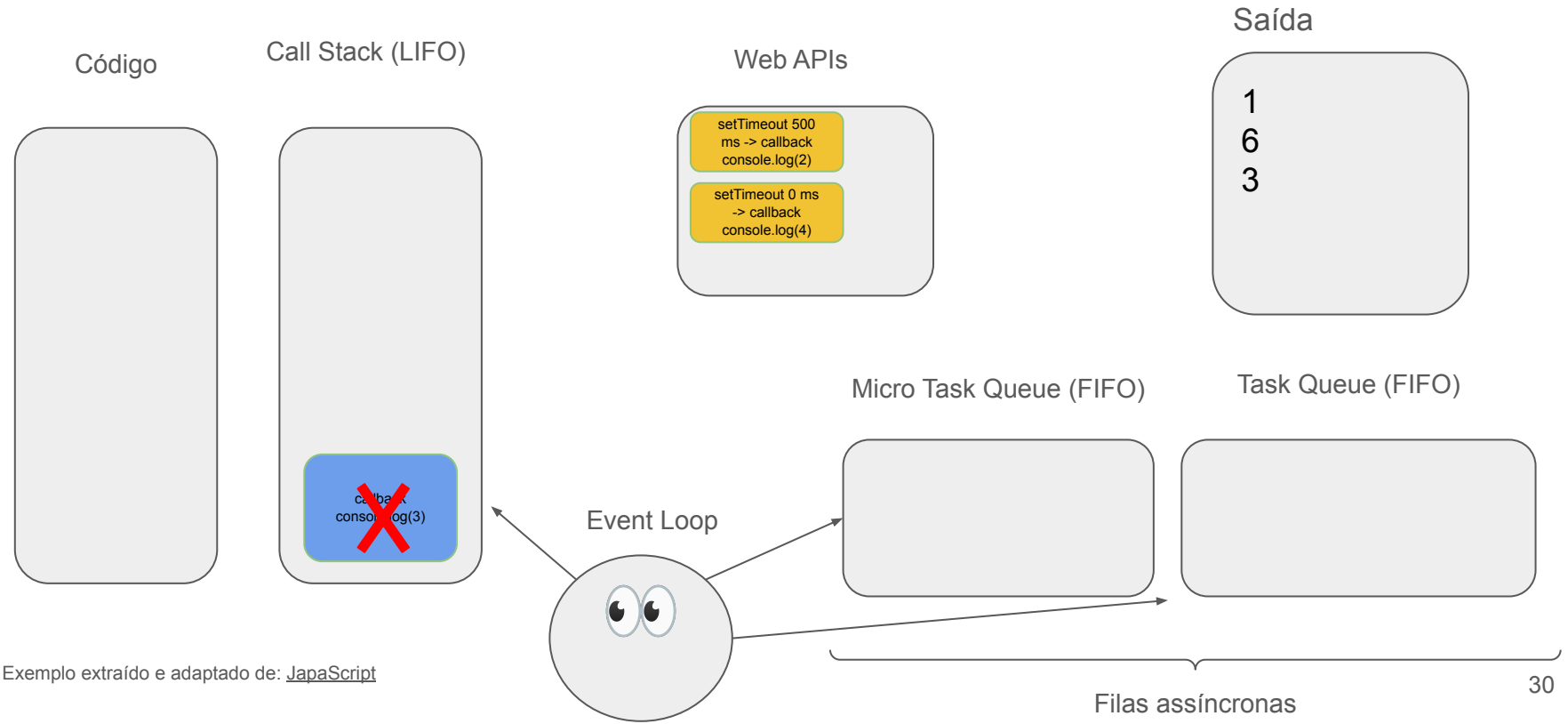
Exemplo extraído e adaptado de: JapaScript

MODELO DE EXECUÇÃO DO JAVASCRIPT



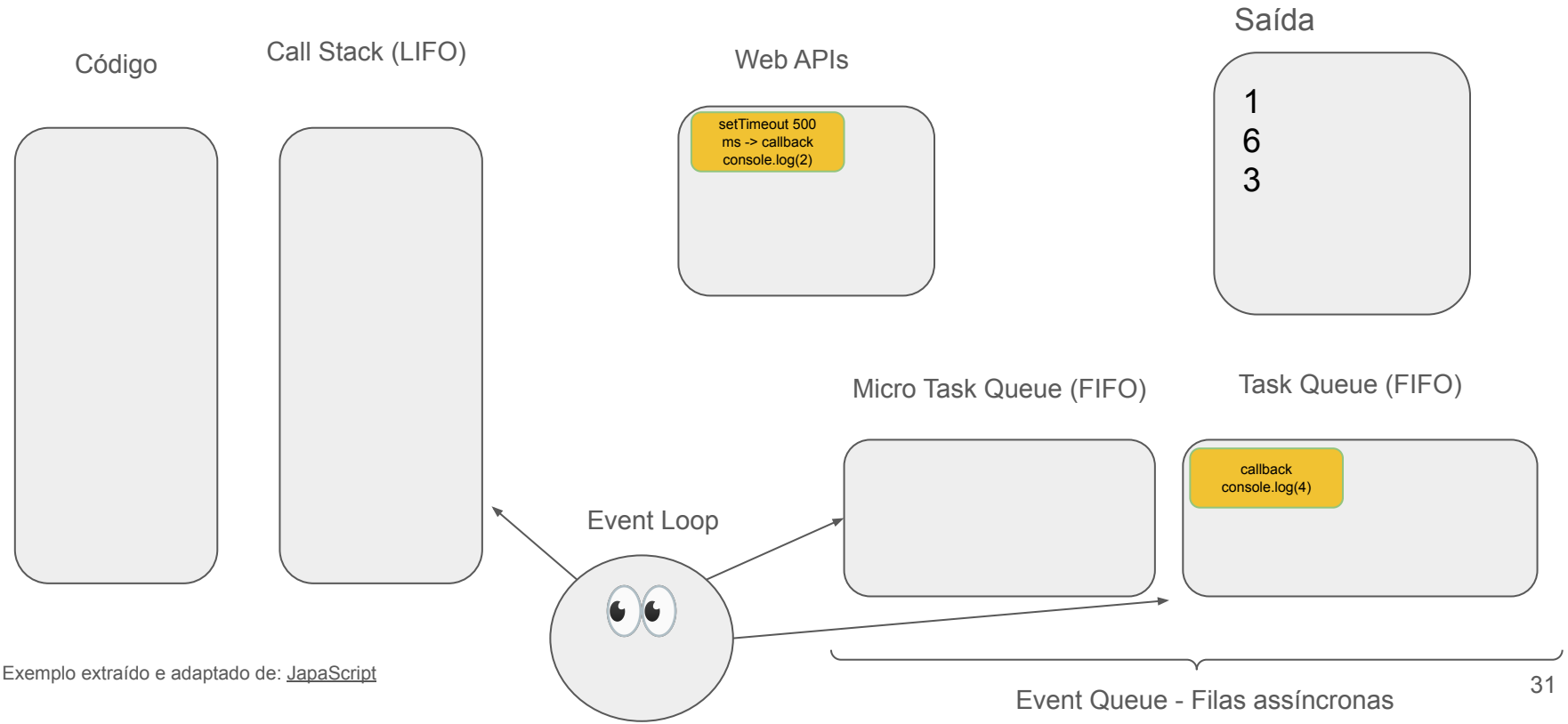
Exemplo extraído e adaptado de: JapaScript

MODELO DE EXECUÇÃO DO JAVASCRIPT



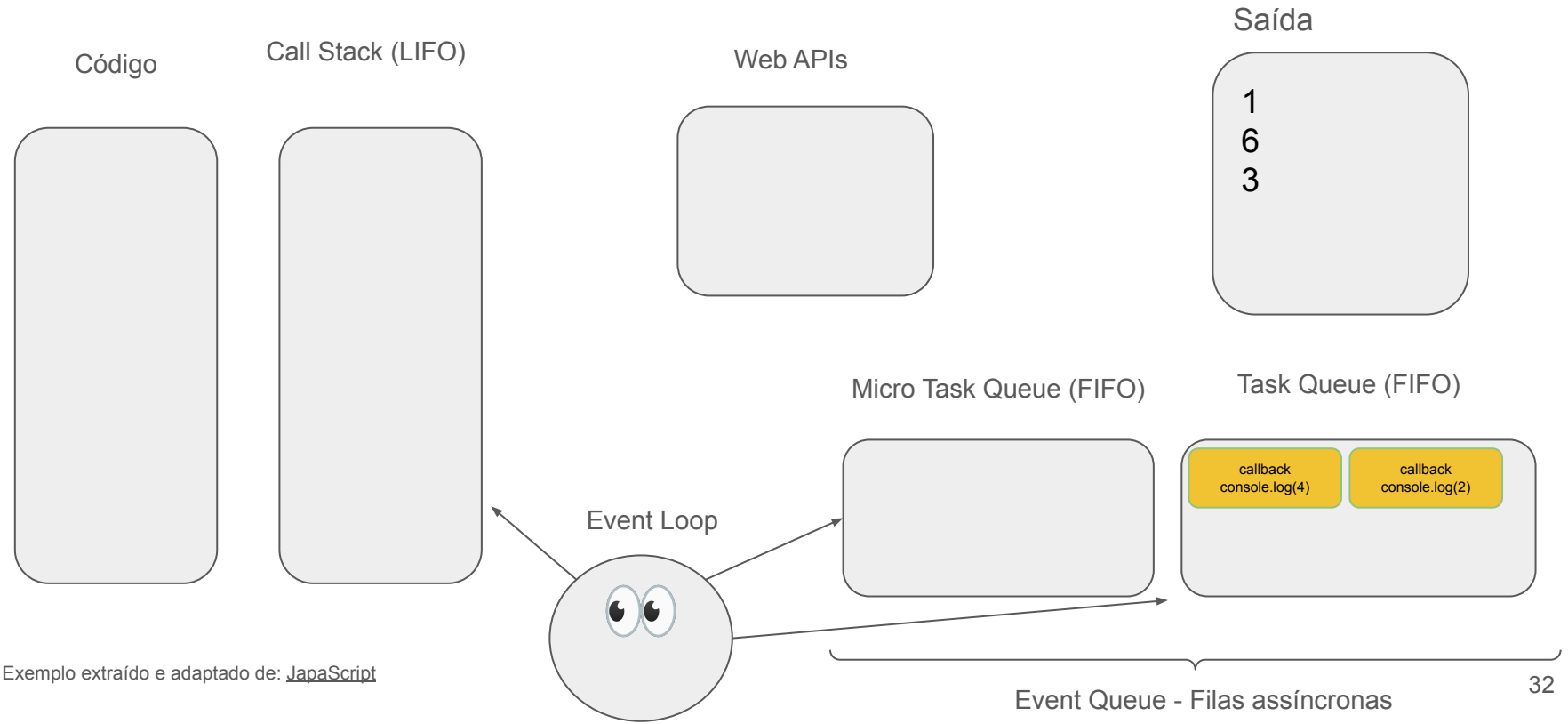
Exemplo extraído e adaptado de: JapaScript

MODELO DE EXECUÇÃO DO JAVASCRIPT



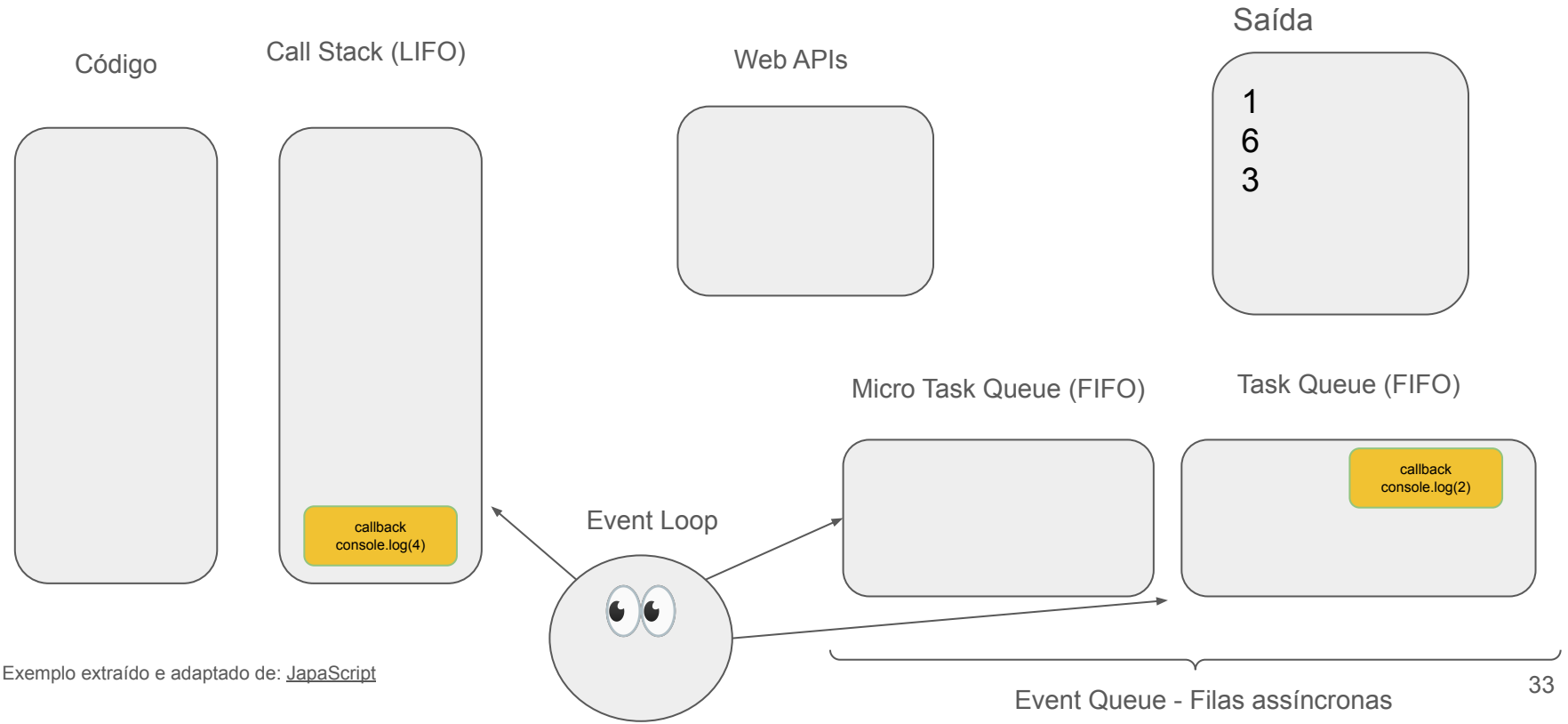
Exemplo extraído e adaptado de: JapaScript

MODELO DE EXECUÇÃO DO JAVASCRIPT



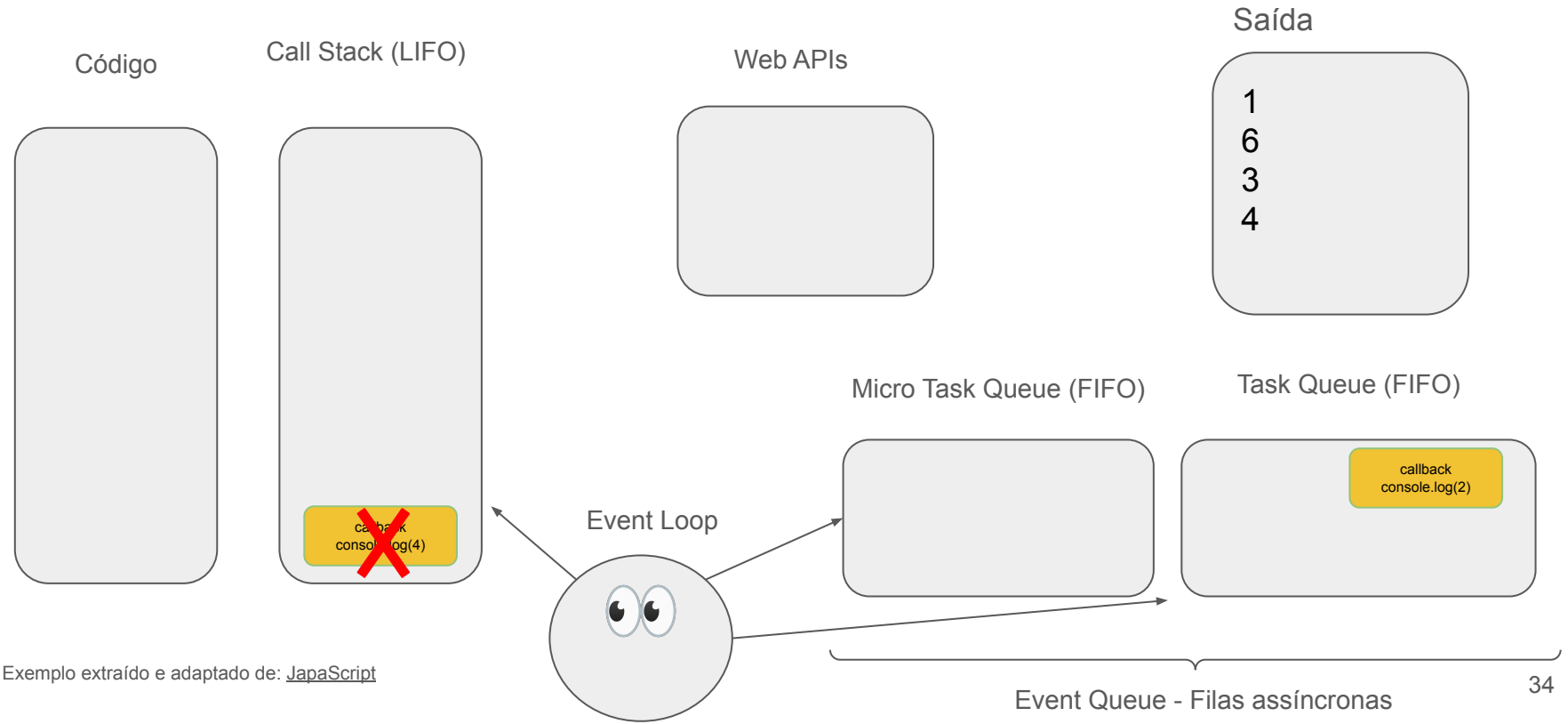
Exemplo extraído e adaptado de: JapaScript

MODELO DE EXECUÇÃO DO JAVASCRIPT



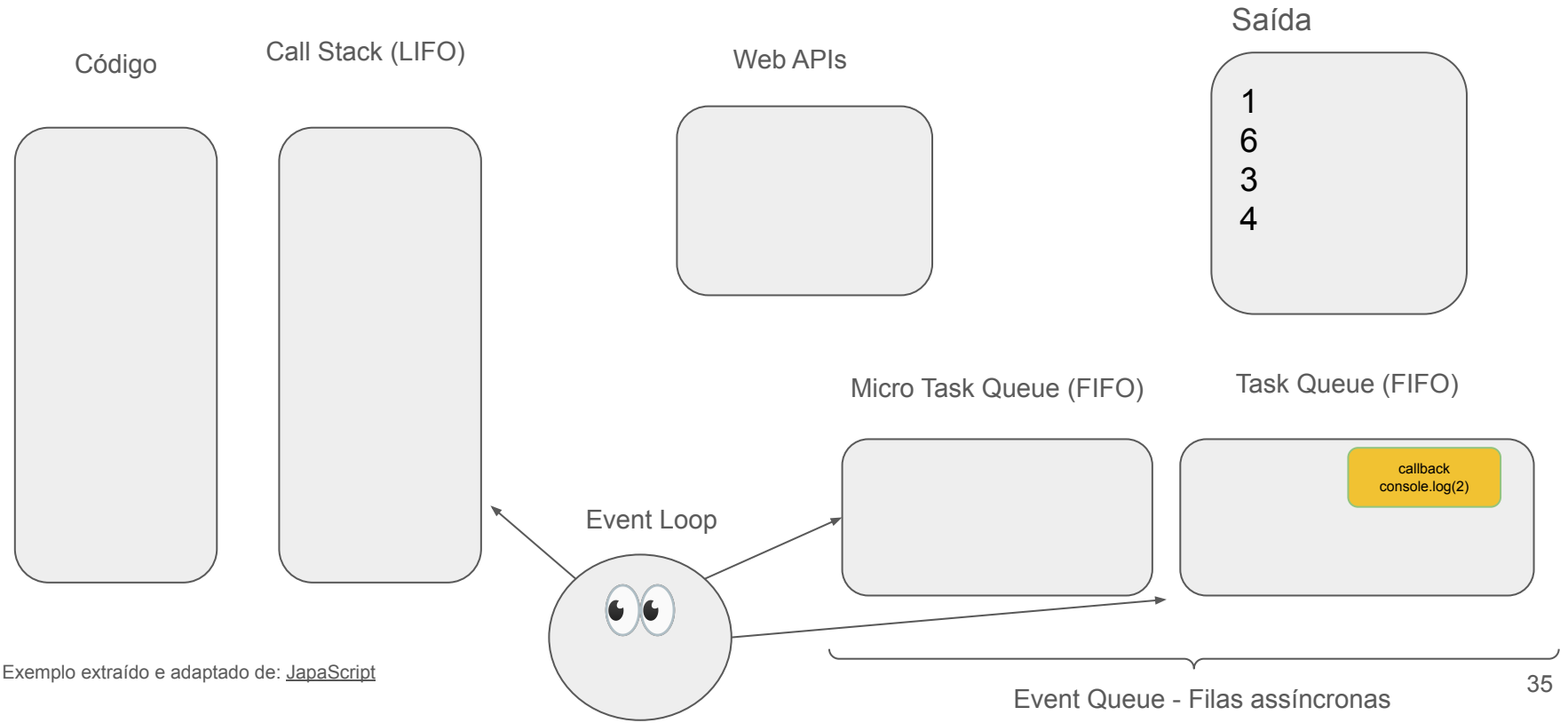
Exemplo extraído e adaptado de: JapaScript

MODELO DE EXECUÇÃO DO JAVASCRIPT



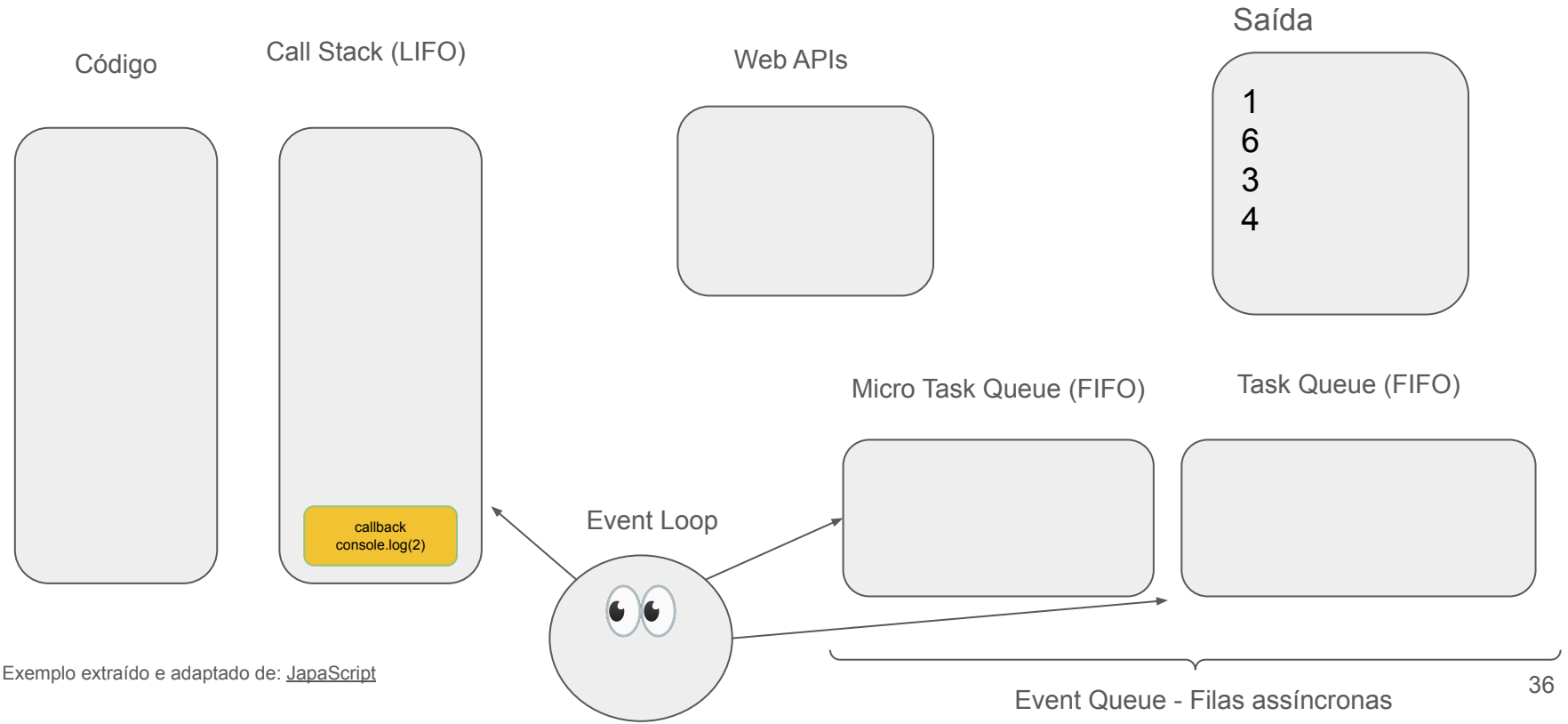
Exemplo extraído e adaptado de: JapaScript

MODELO DE EXECUÇÃO DO JAVASCRIPT



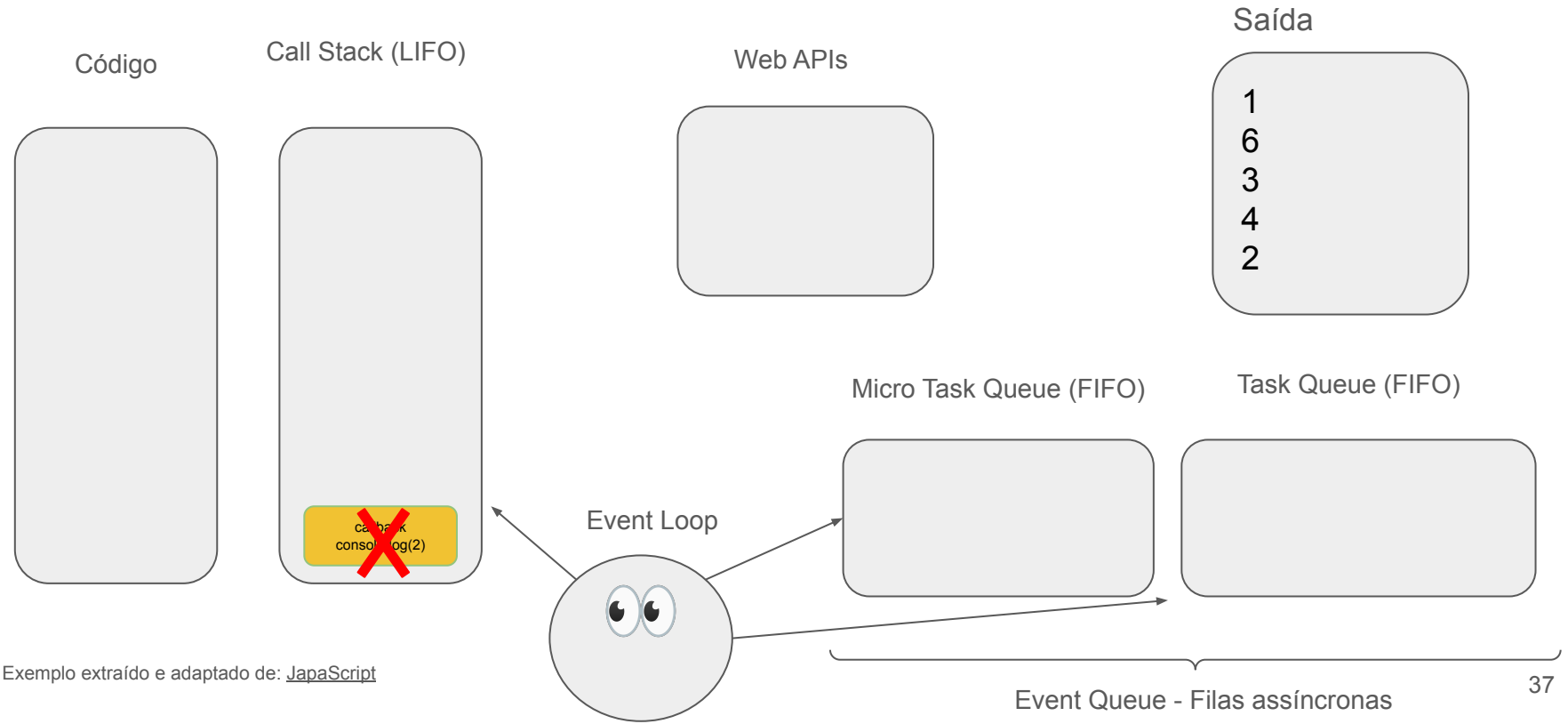
Exemplo extraído e adaptado de: JapaScript

MODELO DE EXECUÇÃO DO JAVASCRIPT



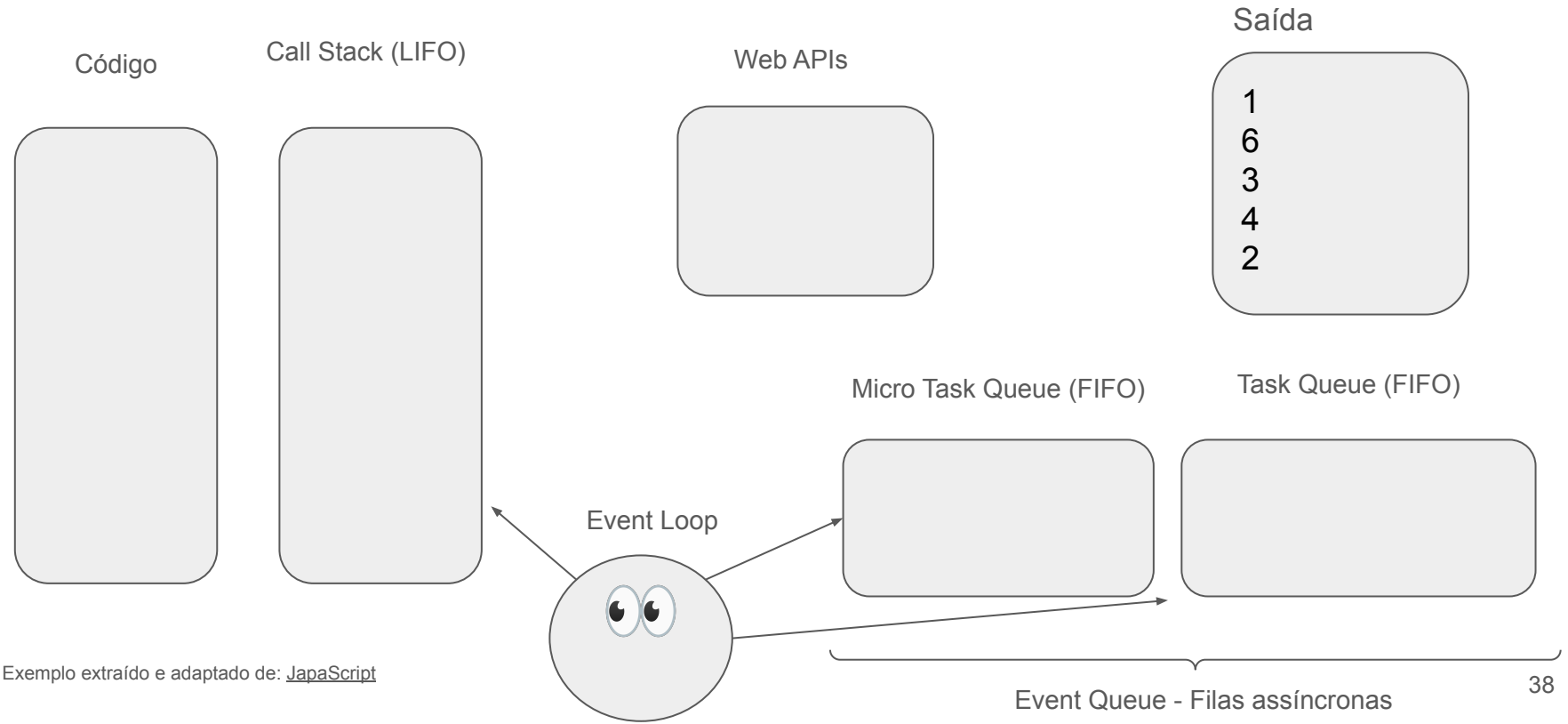
Exemplo extraído e adaptado de: JapaScript

MODELO DE EXECUÇÃO DO JAVASCRIPT



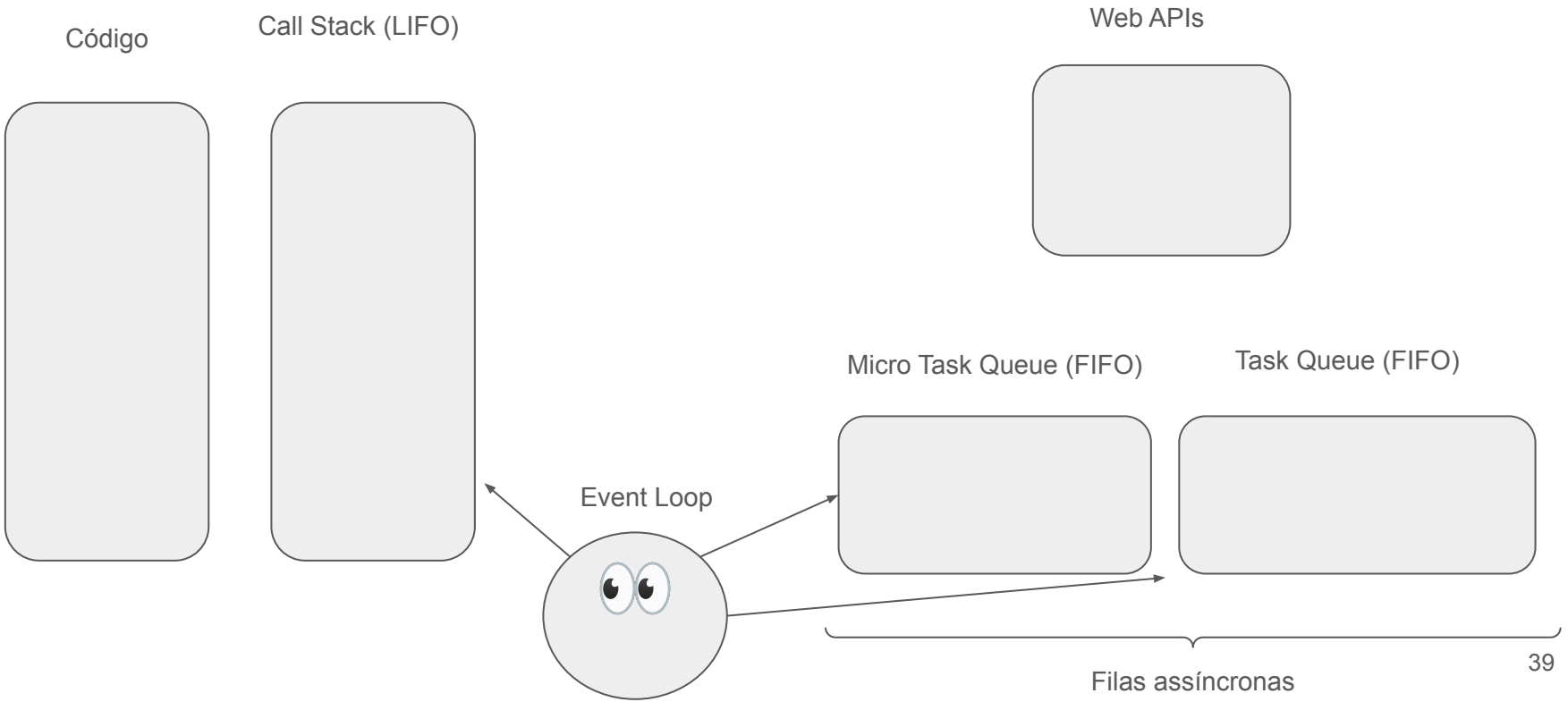
Exemplo extraído e adaptado de: JapaScript

MODELO DE EXECUÇÃO DO JAVASCRIPT

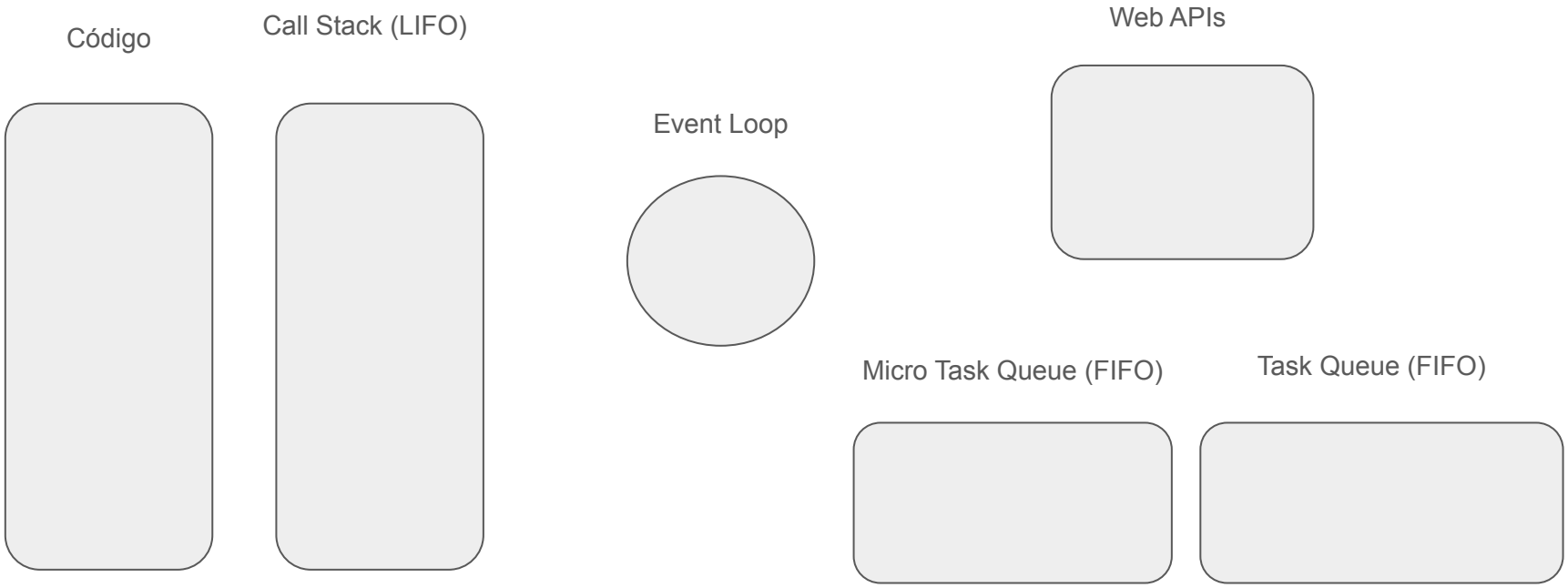


Exemplo extraído e adaptado de: JapaScript

MODELO DE EXECUÇÃO DO JAVASCRIPT



MODELO DE EXECUÇÃO DO JAVASCRIPT



MÃO NA MASSA

MÃO NA MASSA

1. Qual será a saída no console? Explique o porquê.

```
console.log("Antes do setTimeout")

setTimeout(() => {
    console.log("Dentro do setTimeout")
}, 0)

console.log("Depois do setTimeout")
```

MÃO NA MASSA

2. O que será impresso no console?

```
console.log("Início")

setTimeout(() => {
    console.log("setTimeout")
}, 0)

Promise.resolve().then(() => {
    console.log("Promise")
})

console.log("Fim")
```

MÃO NA MASSA

3. O que será impresso no console?

```
console.log("1")

setTimeout(() => {
    console.log("2")
}, 0)

Promise.resolve().then(() => {
    console.log("3")
})

console.log("4")
```

MÃO NA MASSA

4. O que será impresso no console?

```
const fetchPokemonData = async () => {
    try {
        const result = await fetch('https://pokeapi.co/api/v2/pokemon/ditto' )
        const data = await result.json()
        console.log("4")

    } catch (error) {
        console.log(error)
    }
}

console.log("1")

setTimeout(() => {
    console.log("2")
}, 0)

Promise.resolve().then(() => {
    console.log("3")
})

fetchPokemonData()

console.log("5")
```

REFERÊNCIAS

- https://developer.mozilla.org/pt-BR/docs/Glossary/Callback_function
- https://youtu.be/3ggEPYvgdb0?si=IkFbL05omEH4d-RF
- latenflip