



# Análise e Tratamento de Erros em JavaScript

☰ Conhecimentos	
📅 Data da aula	@31 de maio de 2024
☰ Tipo	Atividade em Classe Aula Expositiva

## Objetivos de Aprendizagem

1. Aprender sobre Alocação de Memória
2. Aprender sobre Stack e Heap
3. Aprender sobre Tratamento de Exceções

## Introdução

Uma máxima no desenvolvimento de software é que as coisas darão errado. Nada é mais certo do que isso, principalmente quando começamos a deixar os *Hello Worlds* e **CRUDs** básicos.

Assim, vamos tratar os principais fundamentos da análise e tratamento de erros em JavaScript (também chamado de *Error Handling*), principalmente

falando de *stacktrace* mas incluindo também as estruturas de tratamento (*try/catch/finally*).

## JavaScript - Alocação de Memória

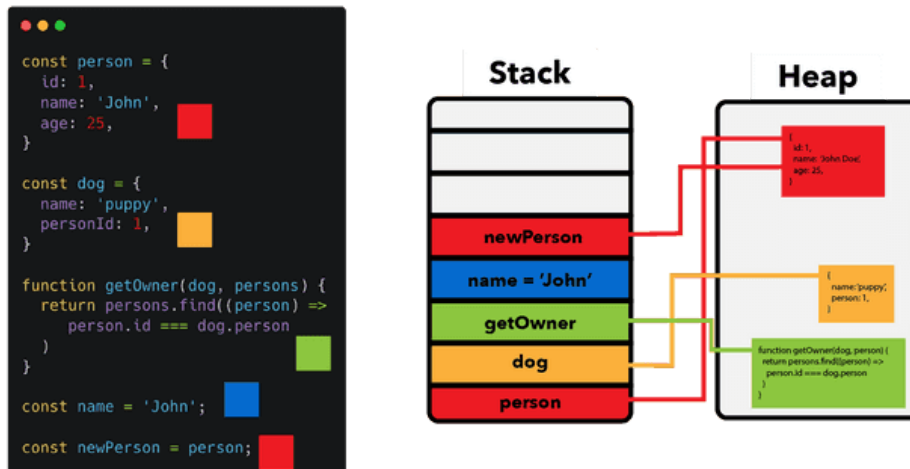
A primeira coisa que você precisa entender, antes mesmo de estudar os erros, é sobre os conceitos básicos de alocação de memória.

Geralmente nossos algoritmos vão ter *inputs*, um processamento deles e *outputs*. Esses *inputs* ou dados de entrada precisam ser armazenados em memória antes do processamento ocorrer, momento onde serão acessados e, mais tarde, após o processamento, serão liberados. Para que todos esses *inputs* não se percam na memória do computador, quando um programa é executado, o *runtime* aloca duas áreas de memória chamadas de *heap* e de *stack*, responsável por manter tudo que o programa precisa e enquanto ele precisar.

A *Heap* é uma área de memória dinâmica, utilizada para alocação de dados de tamanho variável, por exemplo o conteúdo de *arrays*, de funções e o conteúdo de objetos. Funções e objetos podem ter tamanhos extremamente variáveis.

### Stack e JavaScript

Já a *Stack* é uma área de memória estática, utilizada para alocação de dados de tamanho fixo como as primitivas do JavaScript (*number*, *string*, etc) e referências para funções e objetos. Repare que em ambas citamos objetos e funções, mas enquanto na primeira falamos de conteúdo, na segunda falamos de referência. Uma referência de memória (ou ponteiro/*pointer*) é apenas o registro na *stack* de um local na *heap*, como mostra a imagem abaixo, onde a variável de referência "name" na *stack* aponta para um *array* de objetos na *heap*.



Exemplo demonstrativo - Stack vs Heap

Dessa forma, podemos entender a *stack* como um índice da *heap*, mas tem mais algumas coisas sobre ela precisamos saber, sendo a principal, como que sua alocação se dá, sendo que ela não possui o nome de "*stack*" à toa.

*Stack* significa "pilha" e esse nome vem em alusão às pilhas do mundo real, onde colocamos um objeto em cima do outro como em uma pilha de pratos ou de livros. Então conforme você vai declarando suas variáveis, seus valores (no caso de primitivas) ou suas referências (no caso de objetos e funções), vão sendo empilhados na memória *stack*, um em cima do outro.

Mais tarde, conforme as funções terminam e as variáveis não são mais necessárias, inicia-se o processo de "desempilhar", ou seja, vamos removendo da *stack* as referências e variáveis, uma a uma, seguindo a ordem LIFO ou Last In, First Out (o último a entrar, é o primeiro a sair). Quando o último elemento é removido e a *stack* fica vazia, o programa é encerrado.

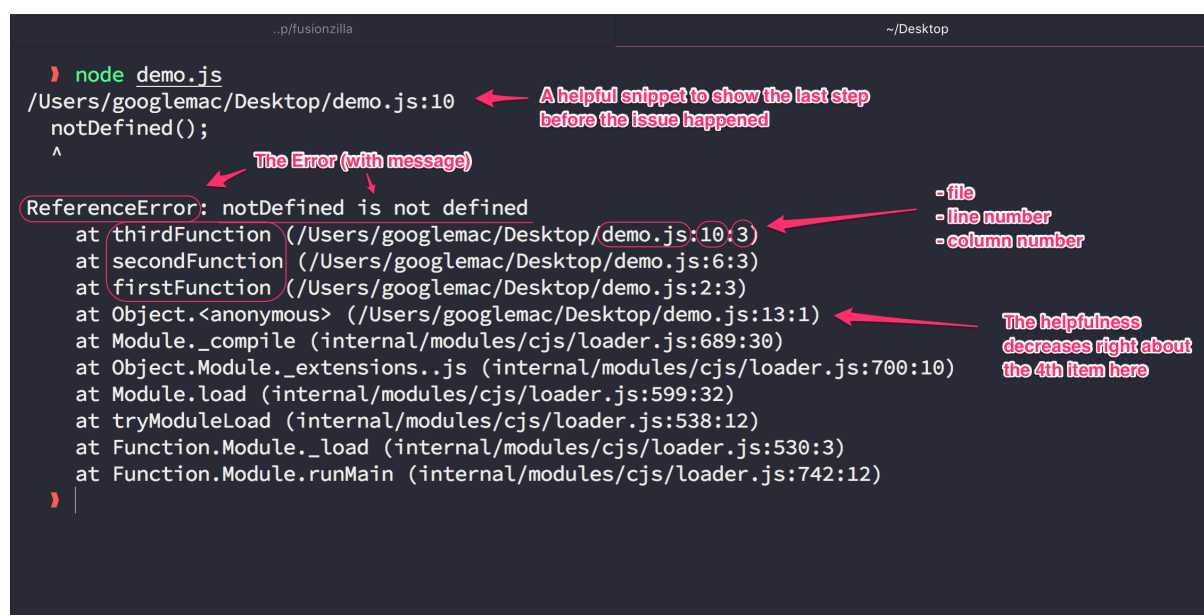
Mas e se algo dá errado durante o processamento?

## JavaScript – Stack Trace

Quando um comportamento inesperado acontece, é o que chamamos de exceção. As exceções são objetos de erro enviados pelo *runtime* e que interrompem o fluxo natural do processamento caso não sejam tratados. Junto à mensagem da exceção é incluído sempre o rastro da pilha ou *stack trace*. E é

na análise desses dois elementos que você vai se basear para entender o que houve e poder tratar ou solucionar a exceção mais tarde.

O *stack trace* nada mais é do que um resumo do estado da stack no momento da exceção, ou mais simplesmente: as últimas funções chamadas antes do erro acontecer, na ordem inversa em que foram executadas (é uma pilha, lembra?). A parte boa é que além de te dar o nome da função, o rastro da pilha te fornece o caminho completo até o arquivo, linha e coluna onde a mesma foi disparada.



```
node demo.js
/Users/googlemac/Desktop/demo.js:10
  notDefined();
  ^
ReferenceError: notDefined is not defined
    at thirdFunction (/Users/googlemac/Desktop/demo.js:10:3)
    at secondFunction (/Users/googlemac/Desktop/demo.js:6:3)
    at firstFunction (/Users/googlemac/Desktop/demo.js:2:3)
    at Object.<anonymous> (/Users/googlemac/Desktop/demo.js:13:1)
    at Module._compile (internal/modules/cjs/loader.js:689:30)
    at Object.Module._extensions..js (internal/modules/cjs/loader.js:700:10)
    at Module.load (internal/modules/cjs/loader.js:599:32)
    at tryModuleLoad (internal/modules/cjs/loader.js:538:12)
    at Function.Module._load (internal/modules/cjs/loader.js:530:3)
    at Function.Module.runMain (internal/modules/cjs/loader.js:742:12)
```

Annotations in the image:

- A helpful snippet to show the last step before the issue happened (points to line 10 of demo.js)
- The Error (with message) (points to the ReferenceError message)
- file  
- line number  
- column number (points to the file path and line/column numbers in the stack trace)
- The helpfulness decreases right about the 4th item here (points to the 4th item in the stack trace)

## Entendendo o Tratamento de Exceções

O tratamento de exceções é um recurso fundamental em muitas linguagens de programação, incluindo JavaScript, que permite ao desenvolvedor gerenciar erros de forma controlada. Em JavaScript, isso é realizado por meio dos blocos `try`, `catch`, `finally` e da instrução `throw`.

### O bloco `try catch`

O bloco `try` é usado para envolver o código suscetível a erros. O JavaScript tenta executar o código dentro deste bloco, e se um erro ocorre, a execução é interrompida e o controle é transferido para o bloco `catch` mais próximo.

```
try {
  // seu código aqui
```

```
} catch (error) {  
    // tratamento de erro aqui  
}
```

## O Bloco **finally**

Você também pode estender o try catch usando a cláusula **finally**. Este bloco será executado independente se houver ou não falha, ou seja, depois que o try ou catch executar, este bloco será acionado. Isto pode ser útil por exemplo, para fechar um arquivo que foi aberto para leitura, registrar algum log ou fechar alguma conexão. Veja a sintaxe de exemplo.

```
try {  
    // seu código aqui  
} catch (error) {  
    // tratamento de erro aqui  
} finally {  
    // executa sempre  
}
```

## A instrução **throw**

Para lançar uma exceção manualmente, utiliza-se a instrução **throw**. Esse recurso permite que o desenvolvedor crie condições específicas de erro, as quais podem ser capturadas e tratadas pelos blocos **catch**.

### Exemplo Prático:

```
function verificarIdade(idade) {  
    try {  
        // Suponha que exista uma regra que a idade deve ser maior  
        if (idade < 18) {  
            // Lançando uma string como "erro"  
            throw "Desculpe, você deve ter mais de 18 anos.";  
        }  
  
        // Se a idade for maior ou igual a 18, essa mensagem será  
        console.log("Idade verificada com sucesso.");  
    }  
}
```

```
} catch (erro) {  
  // Capturando o "erro" lançado e exibindo a mensagem  
  console.log("Erro: " + erro);  
}  
finally {  
  // Este bloco será executado independentemente do resultado  
  console.log("Verificação de idade concluída.");  
}  
}  
  
verificarIdade(16)
```

No exemplo acima, a instrução `throw` é usada para lançar um erro manualmente, que é então capturado pelo bloco `catch`. Independentemente do que aconteça nos blocos `try` e `catch`, o bloco `finally` será executado.

Este mecanismo de tratamento de exceções permite que você escreva aplicativos mais robustos e confiáveis, tratando erros de maneira controlada sem interromper completamente a execução do programa.

## O Objeto `Error` do JavaScript

O objeto `Error` é uma peça fundamental no tratamento de exceções em JavaScript. Ele é utilizado para criar uma instância de erro que pode ser lançada com a instrução `throw` e posteriormente capturada por um bloco `catch`. O objeto `Error` não só contém a mensagem de erro como também outras propriedades úteis, como o nome do erro e a pilha de chamadas, que podem ser extremamente valiosas para depuração.

### Propriedades e Métodos

- **message:** Uma descrição da mensagem de erro.
- **name:** O nome do erro (por exemplo, `TypeError`, `ReferenceError`).
- **stack** (não padrão): Uma representação da pilha de chamadas no momento em que o erro foi lançado.

### Criando e Lançando um `Error`

Para criar um erro, simplesmente instancie o objeto `Error` com a mensagem de erro desejada:

```
const meuErro = new Error("Mensagem de erro personalizad  
a");  
throw meuErro;
```

Este erro personalizado pode ser capturado usando um bloco `catch`:

```
try {  
  throw new Error("Mensagem de erro personalizada");  
} catch (error) {  
  console.log(error.message); // Saída: Mensagem de erro pe  
rsonalizada  
}
```

O tratamento de exceções em JavaScript é uma ferramenta poderosa que permite aos desenvolvedores gerenciar erros de maneira eficaz. Ao entender e utilizar os blocos `try`, `catch`, `finally`, a instrução `throw` e o objeto `Error`, é possível escrever códigos mais robustos e confiáveis, melhorando significativamente a qualidade das aplicações JavaScript.

## Atividades de Aprendizagem

1. Escreva uma função JavaScript que receba um número como parâmetro e gere um 'Erro' personalizado se o número não for um número inteiro.
2. Escreva um programa JavaScript que use um bloco try-catch para capturar e tratar um 'TypeError' ao acessar uma propriedade de um objeto indefinido.
3. Escreva uma função JavaScript que aceite dois números como parâmetros e gere um 'Erro' personalizado se o segundo número for zero.
4. Escreva uma função JavaScript que receba um número como parâmetro e gere um 'Erro' personalizado se o número for negativo.
5. Escreva uma função JavaScript que receba um array como parâmetro e gere um 'Erro' personalizado se o array estiver vazio.
6. Escreva uma função JavaScript que receba uma string como parâmetro e gere um 'Erro' personalizado se a string estiver vazia.

7. Escreva um programa JavaScript que use um bloco try-catch para capturar e tratar um 'RangeError' ao acessar um array com um índice inválido.
8. Escreva um programa JavaScript que demonstre o uso da instrução 'try-catch-finally' para detectar e tratar um erro e, em seguida, execute algum código de limpeza no bloco 'finally'.

## Atividades Extras

### Exercício 1 - Cálculo de IMC

Criar um código para calcular o Índice de Massa Corporal, recebendo dados de altura e peso.

Fórmula:

$$IMC = peso \div altura^2$$

### Exercício 2 - Área e perímetro de retângulo

Receba ou armazene os valores de lado de um retângulo e exiba os valores de área e perímetro.

### Exercício 3 - Verificação de divisibilidade

Receba ou armazene dois valores ( `x` e `y` ) e informe se `x` é divisível por `y` (a divisão dá resto zero).

### Exercício 4 - Comparação de números

Crie um programa que compara dois números e informa se o primeiro é maior, menor ou igual ao segundo.

Utilize **if/else**.

### Exercício 5 - Calculadora

Faça uma calculadora básica, que recebe dois números e a operação entre eles, e retorne o resultado.

Utilize o **switch** para armazenar a operação escolhida. Utilize um array ou um objeto para servir como input dos dois número e a operação desejada.