**História e Conceitos**

**JavaScript**: Lançado em setembro de 1995. **Moca**, **livescript** e por fim **JavaScript.**

**Criado por:** Brendan Eich

**ECMAScript:** “S6, S7”. É uma especificação de uma linguagem de programação que é padronizada pelo **ECMA262,** servindo como base para outras linguagens **script,** como **JavaScript**, **JScript** e **EctionScript.**

**TC39:** É o comitê responsável pela evolução JavaScript. Ele vem para manter e atualizar os padrões do **ECMAScript** que é no projeto **ECMA262**, ele identifica, desenvolve e mantem padrões para as bibliotecas que estende nas **fitures** do **ECMAScript** que é o **ECMA402.** Ele desenvolve **suifts** de testes para que as propostas podem ser testadas, esse projeto se chamada **ECMATR104.**

**Fluxo da Proposta:**

**- Stage 0: strawman:** Vai submeter um formulário com a ideia de evoluir o **ECMAScript**, deve ser feita por um membro ou contribuidor registrado no **TC39.**

**Obrigatorio:** Documento revisado no encontro da **TC39** e depois adicionados as pages da proposta como stage 0.

**- Stage 1: proposal:** Surge a proposta formal da funcionalidade.

**Obrigatório:** Vai ter um **Champion** e ele precisa ser membro do **TC39.** Bem descritos os objetivos da proposta e exemplos.

**- Stage 2: draft:** Primeira versão dessa proposta que vai entrar na verificação.

**Obrigatório:** A proposta precisa ter uma descrição formal da sintaxe e semântica da funcionalidade, a descrição precisa ser o mais completa possível. Duas implementações são necessárias dessa proposta, sendo que uma delas pode rodar pelo BABEL.

**- Stage 3: canditate:** Feedback de implementação e de usuário super ID, especificação completa nesta etapa, assinatura do editor.

**Obrigatório:** Deve ter pelo menos duas especificações compatíveis com a implementação.

**- Stage 4: finished:** Pronta para ser incluída na especificação, mas ainda tem alguns pontos para progredir.

Obrigatório: Estar passando na suwit de testes, duas implementações de conformidade com especificação que passa nos testes, experiencia prática significativa na implementação, baseada no feedback anterior e o Editor de especificação do ECMAScript deve assinar o texto dessa esécificação.

**ES2018:** Última especificação que temos

- Operadores rest/spread

- Alteração assíncrona

- Promisse.prototype.finally()

**ES.Next:** Especificação que possui futuras implementações

**Linguagem interpretada:**

O JavaScript é uma linguagem interpretada, ou seja, o código é executado de cima para baixo e o resultado dessa execução do código é imediatamente retornado

**Linguagem de tipagem fraca e dinâmica:**

**Tipagem dinâmica:** não é necessária especificar o tipo da variável na sua declaração. Por exemplo: var número.

**Tipagem estática:** é necessário especificar o tipo da variável na sua declaração. Por exemplo: int número.

**Fraca:** Não a verificação em todas as operações no JavaScript, ou seja, é possível utilizar o operador “+” com uma string e um número sem que ocorra um erro.

**Forte:** retorna um erro caso utilize o operador “+” com uma string e um inteiro.

**TypeScript:**

Super7 do JavaScript. Além de adicionar tipos de desenvolvimento para o JavaScript, ele adiciona funcionalidades que o JavaScript não tem por padrão.

**Flow:**

Semelhante ao TypeScript com relação a checagem de tipos, mas ele não é um Super7 da linguagem. Como se fosse uma versão ais simplificada do TypeScript.

**Funções de primeira classe e ordem maior:**

**Função de primeira classe:** Significa que a função pode ser atribuída a uma variável, estrutura de dados (object ou array) e ela pode ser passada por argumento.

**Closure**:

A capacidade de uma função lembrar do ambiente em que ela foi criada

**Currying: para cada parâmetro será criado uma função**

É a técnica de transformar uma função com diversos parâmetros em apenas uma função que recebe apenas um parâmetro e para cada parâmetro a gente vai retornando uma função.

**Hoisting: significa levantar ou suspender algo**

**Pode ser separado em dois tipos: “variáveis” e “função”**

**Variável:** só eleva a criação da variável e não da função, se a variável for criada depois da chamada, você recebera um undefined, significa que “a variável foi criada, mas não foi encontrada”.

**Função:** ela é elevada ao topo como um todo, pode ser criada/declarada e atribuída depois da chamada da função e mesmo assim ela será executada normalmente.

**Imutabilidade: depois da variável ser criada “P1” não poderá mudar seus valores.**

Deverá criar outra variável “**P2**” para manipular os valores, sem alterar definidamente os valores da “**P1**”.

**Tipos e Variáveis**

**Obs: Antigamente o JavaScript só possuía dois tipos de escopo sendo eles: “Escopo global” e “Escopo de Função”. O “Escopo de bloco” veio junto com o “let” e “const”.**

**Var: Não respeita(entende) o “escopo de bloco”. Só vai aceitar “Escopo de função” e “Escopo global”.**

**Let: Respeita o “Escopo de bloco”**

**Const: Respeita o “Escopo de bloco”**

**Const:** Se for uma **const** eu não consigo alterar a propriedade, só é possível ser alterada se for um **objeto** ou **array**, mas eu não consigo mudar para onde ele está apontando.

**{ }:** Escopo de bloco

**var escopo:** Escopo global

**Function test( ){**

**}:** Escopo de função

**Tipos, variáveis, operadores, condicionais e repetição**

**O javascript possui 6 tipos permitidos de variáveis:**

String, number, boolean, null, undefined, symbol, object, function, array.

**Undefined:** A variável existe mais ainda não foi setada (Não foi atribuído um valor para ela).

**Symbol:** Nos permite criar valores únicos.

**Operadores**

**Aritméticos:**

**Atribuição:  
Comparação:**

**Condicional:  
Lógicos:**

**Spread:**

**Estruturas condicionais e repetição**

**If, else e else if:  
switch:**

**REPETIÇÃO:**

**For:**

**While:  
do...while:**

**For...in:**

**Continue:  
break:**

**--------------------------------------------------------------------------------------------------------------------------------------------------------------**

**Orientação a objetos**

**Herança:**

**-** Baseada em protótipos

**- prototype:** uma variável que armazena as definições do objeto.

**- \_proto\_:** aponta para o prototype do tipo que foi criado

**- contructor:** Baseado no constructor e criado o prototype e nessa variável é armazenado a referência dela pelo \_\_proto\_\_

**Obs:** Toda função construtora tem um **prototype** atrelado a ela e toda vez que é criado uma variável a partir de uma função construtora, terá uma referência do **\_\_proto\_\_** nela.

**“O \_\_proto\_\_ aponta para um prototype e esse prototype é criado a partir de um constructor de uma função construtora.”**

**New Foo (...);**

O que ocorre?

1 – Um novo objeto é criado, herdando Foo.prototype

2 – A função construtora Foo é chamada com os argumentos especificados e com o “this” vinculado ao novo objeto criado.

3 – Caso a função construtora tenha um retorno explícito, será respeitado o seu `return`. Senão, será retornado o objeto criado no passo 1.

**Classes:**

- Ele foi criado no S6

- Simplificação da herança de protótipo

- Palavra-chave **class.**

**Modificadores de acesso:**

**Obs:** Atualmente o Javascript não possui

- Controle do que é privado ou público dentro da classe.

**Encapsulamento:**

**-** Ocultar detalhes do funcionamento interno

**Static: sintaxe “static Walk ()”**

**-** Acessar métodos/ atributos sem instanciar

**Design patterns**

**Definição:**Designe patterns ou padrões de projeto são soluções generalistas para problemas recorrentes durante o desenvolvimento de um software. Não se trata de um framawork ou código pronto, mas de uma definição de alto nível de como um problema comum pode ser solucionado.

**Surgiu no livro “A Pattern Lamguege”**

**-** Em 1978

- Por Christopher Alexander, Sara Ishikawa e Murray Silverstein

- Possui mais de 253 tipos de problemas/desafios e projetos

**Definição do “Formato de um Pattern”**

- Ele deve ter um **NOME**

- **EXEMPLO** na sua utilização

- **CONTEXTO** em que deve ser aplicado

- **PROBLEMA** que esse pattern resolve

- **SOLUÇÃO** a forma com que ele resolve

**Palestra: Using Pattern Language for Object-Oriented Programs**

- Em 1987

- Por Kent Beck e Ward Cunningham

- 5 padrões de projeto

**Livro: Design Patterns: Elements of Reusable Object-Oriented Software**

- Em 1994

- Gang of four (GoF)

- Por Erich Gamma, Richard Helm, Ralph Johnsom, e John Vlissides

**Nesse livro os Designd Patterns são caracterizados em 3 tipos:**

**- Criação**

**- Estrutural**

**- Comportamental**

**Padrões de criação:**

Os padrões de criação são aqueles que abstraem e/ou adiam o processo de criação dos objetos. Eles ajudam a transformar um sistema independente de como seus objetos são criados, compostos e representados.

**- Abstract Factory**

**- Builder**

**- Factory Method**

**- Prototype**

**- Singleton**

**Padrões estruturais:**

Os padrões estruturais se preocupam com a forma como classes e objetos são compostos para formar estruturas maiores.

**- Adapter - Business Delegate**

**- Bridge - Flyweigth**

**- Composite - Proxy**

**- Decorator**

**- Facade**

**Padrões comportamentais:**

Os padrões de comportamento se concentram nos algoritmos e atribuições de responsabilidades entre os objetos. Eles não descrevem padrões de objetos ou de classes, mas também os padrões de comunicação entre os objetos.

**- Chain of Responsibility - State**

**- Command - Strategy**

**- Interpreter - Template Method**

**- Iterator - Visitor**

**- Mediator**

**- Observer**

**Patteners mais usados no Javascript:  
- Factory - Observer**

**- Singleton - Module**

**- Decorator**

**Factory:**

Todas as funções que retorna um objeto, sem a necessidade de chamá-las com o **new,** são consideradas funções Factory(fábrica).

**Singleton:**

O objetivo desse pattern é criar uma única instância de uma função construtora e retorná-la toda vez em que for necessário utilizá-la.

**Padrão muito popular: JQuery.**

**Decorator:**Uma função decorator recebe uma outra função como parâmetro e entende o seu comportamento sem modificá-la explicitamente.

**Pode ser usado com: TypeScript.**

**Observer:**

É um pattern muito popular em aplicações Javascript. A instância (subscriber) mantém uma coleção de objetos (observers) e notifica todos eles quando ocorre mudanças no estado.

**Vue**

**RxJs**

**Module:**É um pattern que possibilita organizarmos melhor o nosso código, sem a necessidade de expor variáveis globais.

**----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------**

**Criação e manipulação de Arrays**

**Criando um array:**

**- Const** **arr** = [1, 2, 3]; **sintaxe de criação**

**- Const** **arr2** = new **Array**(1, 2, 3); **sintaxe de instanciação**

**Array.of: criando um array chamando a função of().**Cria uma instância de array a partir do número de parâmetros informado. Pode ser número ou string.

**- Const** **arr** = **Array.of(1, 2, 3); sintaxe**

**Array():**  
Cria uma nova instância de array de acordo com os parâmetros informados

**Array.from():**

Cria uma instância de array a partir de um parâmetro **array-like** ou **interable object**

**Push:**

Adiciona um ou mais elementos no final do array e retorna o tamanho do novo array

**Pop:**

Remove o último elemento de um array e retorna o elemento removido

**Unshift:**

Adiciona um ou mais elementos no início do array e retorna o tamanho do novo array

**- Const arr = [‘banana’, ‘melancia’, ‘abacate’]**

**- Const arrLength = arr.unshift(‘acerola’);**

**Shift:**

Remove o primeiro elemento de um array e retorna o elemento removido

**Concat:**

Concatena um ou mais arrays retornando um novo array. **Ele é imutável**, ou seja, ele vai pegar os dois arrays e criar outro, não alterando o valor dos originais.

**- Const arr = [1, 2, 3];**

**- Const arr2 = [4, 5, 6];**

**- Const arrList = arr.concat(arr2);**

**Slice:**

Retorna um novo array “fatiando” o array de acordo com início e fim. Não vai alterar o array original.

**Const arr = [1, 2, 3, 4, 5]**

**Arr.slice(0, 2);** // [1, 2]

**Arr.slice(2);** // [3, 4, 5]

**Arr.slice(-1);** // [5]

**Arr.slice(-3);** // [5, 4, 3]

**Splice:**

Altera um array adicionando novos elementos enquanto remove elementos antigos. **Não é imutável**, ou seja, ele vai alterar o array original.

**forEach;**

Iteração de cada item dentro de um array.

**Const arr = [1, 2, 3, 4, 5];**

**Arr.forEach((value, index) =>{**

**Console.log(`${index}: ${value}`);**

**});**

**Map:**

Retorna um novo array, de mesmo tamanho, iterando cada item de um array

**Flat:**

Retorna um novo array com todos os elementos de um sub-array concatenados de forma recursiva de acordo com a profundidade especificada(depth)

**Values:**

Retorna um Array Iterator que contém os valores para cada elemento do array.

**Const arrIterator = arr.values();**

**Entries:**  
Retorna um array iterator que contém os pares chave/valor para cada elemento do array.

**Const arrIterator = arr.entries();**

**Find:**

Retorna o primeiro item do array que satisfaz a condição

**Const firstGreaterThanTwo = arr.find(value => value > 2);**

**findIndex:**

Retorna o índice do primeiro item de um array que satisfaz a condição

**Const firstIndexGreaterThanTwo = arr.findIndex(value => value > 2);**

**Filter:**

Retorna um novo array com todos os elementos que satisfaz a condição

**Const allValuesGreaterThanTwo = arr.filter(value => value > 2);**

**indexOf:**

Retorna o primeiro índice em que um elemento pode ser encontrado no array

**lastIndexOf:**

Retorna o último índice em que um elemento pode ser encontrado no array

**Includes:**

Retorna um booleano verificando se determinado elemento existe no array

**Some:**

Retorna um booleano verificando se pelo menos um item de um array satisfaz a condição

**Const hasSomeEvenNumber = arr.some(value => value % 2 === 0); // true**

**Students.some(students => students.grade >= 7); // true**

**Every:**

Retorna um booleano verificando se todos os itens de um array satisfazem a condição

**Sort:**

Ordena os elementos de um array de acordo com a condição

**Students.sort((current, next) => current.grade – next.grade);**

**Reverse:**

Inverte os elementos de um array

**Join: Transforma em outro tipo de dado**Junta todos os elementos de um array, separados por um delimitador e retorna uma string

**Reduce:**

Retorna um novo tipo de dado iterando cada posição de um array

**Const arr = [1, 2, 3, 4, 5]**

**Arr.reduce((total, value) => total += value, 0); // 15**

**Students.sort((totalGrade, students) totalGrade += students.grade, 0); // 16**

**Students.sort((totalGrade, students) totalGrade += students.grade, 0); / students.length // 5.333 - média**